Software Engineering Institute
Carnegie Mellon University

Extending AADL for Security Design Assurance of Cyber-Physical Systems

Robert Ellison, PhD
Allen Householder
John Hudak
Rick Kazman, PhD
Carol Woody, PhD

December 2015

TECHNICAL REPORT
CMU/SEI-2015-TR-014

CERT Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0003138

Table of Contents

List of Figures

List of Tables

Abstract

Attacks such as the one that compromised the control systems for Iranian centrifuges demonstrate a growing need to improve the design of security in cyber-physical systems. While much of the work on security has focused on coding, many of the common weaknesses that lead to successful attacks are actually introduced by design. This technical report shows how important system-wide security properties can and must be described and validated at the architectural level. This is done through the adoption and use of the Architecture Analysis and Design Language (AADL) and a further extension of it to describe security properties. This report demonstrates the viability and limitations of this approach through an extended example that allows for specifying and analyzing the security properties of an automotive electronics system.

The report begins with a modeling of threats using the Microsoft STRIDE framework and then translates them into attack scenarios. Next, the report describes—as AADL components, relationships, and properties—the architectural structures, services, and properties needed to guard against such attacks. Finally, the report shows how these properties can be validated at design time using a model checker such as Resolute and discusses the limitations of this approach in addressing common security weaknesses.

1 Research Introduction

1.1 Initial Research Plan for Extending AADL to Include Security

Safety-critical verification of cyber-physical systems (CPSs) has benefited from the use of architectural fault-modeling capabilities provided by Architecture Analysis and Design Language (AADL). Architecture-led hazard analysis using mechanisms such as AADL has become an effective capability in safety fault modeling. The cost of successfully addressing safety compliance has been significantly reduced through the use of extensions to AADL that automate safety analysis and produce safety assessment reports to meet recommended practice standards (such as SAE ARP4761) [SAE 1996]. The Software Engineering Institute (SEI) has used AADL to effectively address design verification for the qualities of safety, reliability, and performance [Lewis 2009, Delange 2013, Feiler 2009, AVSI 2015].

Attacks such as Stuxnet[1] demonstrate a growing need to improve the design of security in CPSs [Kelley 2013]. Other researchers have incorporated selected attack scenarios into modeling languages (e.g., OCL [Almorsy 2013], OWL-DL and SWRL [Asnar 2011], and SysML [Ouchani 2011]). Their research explored adding security analysis capabilities into an architecture description language to better record and analyze the design decisions that are relevant to security. The architecture description language AADL was chosen to leverage existing SEI capabilities for safety and reliability analysis and to extend the use of AADL into the security aspects of software assurance. New rules or language extensions would, however, be needed to address the gaps. When reviewing security vulnerability attacks assembled in an Air Force Research Laboratory (AFRL) study [Calloni 2011] (36 attacks covering 632 out of 945 [MITRE 2015] Common Weakness Enumerations [CWEs]), we observed that current descriptions of vulnerabilities focus on coding.

However, many of the important decisions that establish security in a system are made at the architecture level. To build in security from the start, the security coding requirements must be abstracted into the design characteristics and constraints that should be considered in formal modeling to prevent vulnerabilities. We started with the AFRL report [Calloni 2011], using its identified groups of attack patterns based on CWEs to construct desirable design capabilities for at least two critical security capabilities currently missing in AADL: authentication and input validation. Through an appropriately structured analysis, a detailed system architecture design of a CPS can be analyzed using AADL to prevent such types of CWEs.

We divided the work into two tasks as follows:

1. Enhance AADL to incorporate security design assurance. For a set of attack scenarios, we identified the capabilities, constraints, and other design characteristics that reduce the risk of possible attack success.

[1] Stuxnet was a cyber attack that damaged Iranian centrifuges by compromising the commercial control systems that managed them (https://en.wikipedia.org/wiki/Stuxnet).

2. Apply architectural security modeling to CPSs, and develop a case study to show how formal modeling using AADL could be applied to a CPS to improve the security design of the architecture.

1.2 Value in Improved Security for CPSs

The training[2] for AADL provided an automobile example to evaluate the safety features of a proposed design. We found value in exploring this same example to see how security could be addressed in the design. First, because the AADL model was already built, we didn't have to spend time constructing one from scratch. Also, security problems linked to various types of automobiles and based on attacks using external connectivity were frequently reported in the news, indicating the need for strengthening security in this type of CPS. These examples of recent attacks against automobiles have been reported:

- A wireless device used by Progressive Insurance to gather information about customers' driving habits and communicate data to a monitoring station lacked adequate authentication security, so the device could be exploited to unlock car doors, start cars, and access engine information. The device has been used in more than two million vehicles since 2008.[3]
- Add-on devices such as Zubie, a combined hardware and software solution that tracks a car's location and movement and tells drivers how to drive more effectively, also provide hackers the ability to remotely manipulate a car's physical operation and expose driver data.[4]
- Hackers demonstrated how they took control of a Jeep Cherokee through connectivity provided in the entertainment system and successfully disabled the car's transmission and brakes.[5]
- A vulnerability in the OnStar mobile service in GM vehicles allowed hackers to unlock, remotely start, and track those vehicles.[6]
- An attack launched by researchers who installed a Trojan on the Tesla Model S network succeeded in remotely shutting off the car's engine.

These attacks exhibit characteristics of the security failures we wanted to explore. Critical automotive components such as the transmission and brakes are acting on input from untrusted sources (entertainment system or driver-monitoring devices). This research project analyzed mechanisms currently available within AADL, which is a formal modeling language. The goal was to determine how a design could be evaluated to establish confidence that sufficient authentication and input validation are in place to deter the types of attacks cited above.

---

[2] The Modeling System Architectures Using the Architecture Analysis and Design Language (AADL) course (http://www.sei.cmu.edu/training/p72.cfm)

[3] Progressive SnapShot device attack: http://arstechnica.com/security/2015/01/wireless-device-in-two-million-cars-wide-open-to-hacking/ and http://www.scmagazine.com/insurance-dongle-could-be-compromised/article/393707/

[4] Zubie vulnerability: http://www.autelligence.com/car-hacked-telematics-add/

[5] Jeep Cherokee attack: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[6] OnStar vulnerability: http://arstechnica.com/security/2015/07/ownstar-researcher-hijacks-remote-access-to-onstar/

1.3 Audience and Structure of This Report

Readers of this report do not need a background in AADL or formal modeling. However, those wishing to apply the security approaches recommended by the analysis will need this background knowledge.[7]

This report has been prepared to provide those responsible for program management and design review with an understanding of the ways formal modeling can support the verification of key aspects of security design. These readers should focus on the first two chapters and the conclusion. System and software engineers with general knowledge of security issues and architecture will gain an understanding of the ways in which formal modeling with AADL can validate that designs demonstrate key security characteristics. Modeling tools can demonstrate that these evolving designs continue to exhibit the desired security characteristics.

Section 2 provides important background information about modeling security and the aspects of security that can be established and confirmed from an architecture.

Section 3 describes the importance of threat modeling in establishing the desired security properties. The chapter also covers how existing modeling analysis techniques can be applied to establish confidence that security has been properly addressed in a design.

Section 4 describes specific security properties as applied to formal modeling and how design analysis techniques can be applied to establish confidence that security has been properly addressed in a design.

Section 5 describes the AADL model for the automotive example we developed and the details of how the model can be evaluated for specific security properties.

Section 6 provides a summary of the accomplishments and limitations of this research activity and opportunities for further research.

[7] Read about Architecture Analysis and Design Language at http://www.aadl.info.

2 Analyzing Security Risks

Preventing security defects typically depends on analysis that extends beyond a single software component and requires architectural analysis. As the Object Management Group (OMG) noted, detecting back-doors, cross-site scripting vulnerabilities (XSS), or unsecure dynamic SQL queries through multiple layers requires a deep understanding of all the data manipulation layers, as well as the data structure itself [OMG 2013]. System-level analysis allows us to visualize complete transaction paths from user entries, through user authentication and business logic, down to sensitive data access to evaluate the effectiveness of the security design. For example, a weakness in the functional software design can affect access control and authentications and provide a means of bypassing expected controls.

Thus, identifying and mitigating the risks of software design or architectural flaws are essential security analysis activities. Security practices commonly applied for such risk analysis include

- identifying the system features and usage that provide opportunities for an attack (otherwise known as the *attack surface*)
- analyzing how a design could be compromised and the risk of such compromises could be mitigated (otherwise known as *threat modeling*)

We also need to show that security risks have been reduced by demonstrating that architectural decisions have proactively reduced the likelihood or the consequences of a security fault (otherwise known as an *assurance case*).

2.1 Attack Surface

An attacker needs some kind of access to compromise a system, and the earliest exploits often achieved such access by convincing a user to insert a compromised floppy disk. Today, an attacker might try to persuade a user to follow an URL in an email message. Exploring and reducing potential attack opportunities provide a means of improving security [Howard 2003a].

A system's attack surface can be described along three abstract dimensions: (1) targets and enablers, (2) channels and protocols, and (3) access rights [Howard 2003b]. Consider the examples of automotive compromises described in Section 1: The target was the automotive control system. At the most general level, channels include all communication links to the automobile such as OnStar mobile service connectivity, the on-board diagnostic (OBD) port, and external channels provided by entertainment devices such as input from mobile devices. Enablers include the entertainment components and the plug-in devices for the OBD port. The latter, for example, could enable attackers to monitor a motorist's driving behavior via wireless connectivity.

2.2 Threat Modeling

An attack surface simply lists the features that an attacker might try to compromise, and we need to analyze how that compromise might occur. The CWE, developed by the MITRE Corporation,

describes over 900 software weaknesses that have enabled successful attacks.[8] These weaknesses have appeared predominantly in the functional software rather than in the security services such as those providing authentication and authorization capabilities. Compromises typically occur when an attacker creates operating conditions not anticipated by the software designer. For example, a programmer might allocate sufficient memory to accept a city name of up to 200 characters from a user and assume, in practice, that the name provided will always meet that criteria. But the lack of a size check has enabled attackers to gain access by inserting new code into the memory that goes around the authentication and access controls.

Most system development activities seek to provide reliable and trustworthy software, but that can be hard to do. Commercial organizations giving high priority to the development of secure software is a relatively recent change. Microsoft's 2006 publication of the book titled *The Security Development Lifecycle* was a step forward [Howard 2006]. The company had made a decision a few years earlier to address the security concerns raised about its products. Early requirements included code scans to enforce good programming practices and secure programming training. But addressing two tasks during design provides the most significant activity for improving security: (1) analyzing how the proposed software could be compromised and (2) identifying mitigations for high-impact risks. For example, a developer of software for database access should make engineering decisions about how to reduce the risks of known weaknesses associated with such access. This approach seems obvious, but too often developers consider possible security risks only after the code is written. Analyzing how software could be compromised is referred to as *threat modeling*.

There are many ways to do threat modeling. For illustrative purposes in this report, we use the STRIDE[9] approach because it is available through open source documents. Threat modeling uses the following activities to analyze the security risks shown in Table 1:

1. Create information flows for use cases.
2. Gather a list of external dependencies.
3. Define the security assumptions.
4. Determine the threat types.
5. Identify the threats to the system.
6. Determine risk.
7. Plan mitigations.
8. Create external security notes.

[8] http://cwe.mitre.org/community/swa/index.html

[9] https://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx

Table 1: Security Risks

Risk	Description
Data exchanges	Functional requirements include use cases, which describe system behavior in response to an external request such as a data exchange between a user and a system. A data exchange is a security risk factor because a software weakness in its implementation (such as not verifying user input) could provide an adversary with access to business resources such as the customer database.
External dependencies	External systems can be compromised. Exploitable weaknesses can also arise from unknown information about external systems.
Security assumptions	System integration can introduce exploitable mismatches in the security assumptions made by components.

STRIDE, shown in Table 2, focuses on a predefined set of threat types. In Section 4, we apply the STRIDE model to an example CPS in the automotive electronics domain.

An attack surface might include a connection to an external source. The security risks associated with that connection depend on determining the kind of data associated with that connection, identifying the software components that might process data provided by the connection, and analyzing how such specific data could be used to compromise those software components.

Table 2: STRIDE Threat Model

Threat	Security Property
Spoofing	Authentication: the process of determining whether someone or something is, in fact, who or what it is declared to be
Tampering	Integrity: maintaining the consistency, accuracy, and trustworthiness of data over its entire lifecycle
Repudiation	Non-repudiation: Users can't perform an action and later deny performing it.
Information disclosure	Confidentiality: Data access is restricted to those authorized to view the data in question.
Denial of service	Availability: Systems are ready when needed and perform acceptably.
Elevation of privilege	Authorization: the process of adding or denying individual user access to a computer network and its resources

Security assumptions are one type of security risk listed in Table 1. Frequently, an elevation of privilege succeeds because an attack invalidates a security assumption. For example, the design of an organization's internal software systems frequently assumes that an adversary cannot access the internal networks. An attack that compromised the security vendor RSA started by convincing an RSA employee to click on a link to an intruder-created video in an email message.[10] The video was designed to exploit a vulnerability in a third-party extension to the browser and give the intruder access to the RSA internal network as that employee. The intruders then took advantage of the implicit trust in RSA insiders assumed during internal software development to eventually obtain sensitive information.

[10] The RSA Hack: How They Did It: http://bits.blogs.nytimes.com/2011/04/02/the-rsa-hack-how-they-did-it/?_r=0

Threat modeling typically uses data flow diagrams (DFDs) to graphically represent a system. DFDs use a standard set of symbols consisting of four elements:

1. data flows (data in motion)
2. data stores (data at rest)
3. processes (computation)
4. interactors (Endpoints of a system are typically providers or consumers of data.)

Figure 1 shows a high-level DFD for an information system.

Figure 1: Data Flow Diagram for an Information System

A generic DFD for an automotive CPS appears in Figure 2.

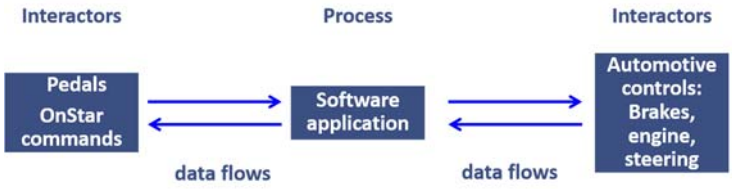

Figure 2: Data Flow Diagram for Automotive System

Figure 3 shows the DFD of a cruise control (CC) system designed in the 1980s. Information was typically exchanged among vehicle controllers with the Controller Area Network (CAN) that was specially developed in the early 1980s for fast serial data exchange between electronic controllers in motor vehicles. Instigators include buttons that the driver can use to turn on the CC system and change its settings. An example of a use case is going off cruise control when the driver presses the brake pedal. The CC control software monitors the frequency of wheel rotations to calculate vehicle speed and sends instructions to the engine controller to increase or decrease the velocity. In a properly designed CC system, the engine controller should only receive input from the CC software component. At the time of the CAN's development, only motorists and mechanics had access to critical vehicle controls, and the CC system was essentially a self-contained one with no attack surface and no need for threat modeling.

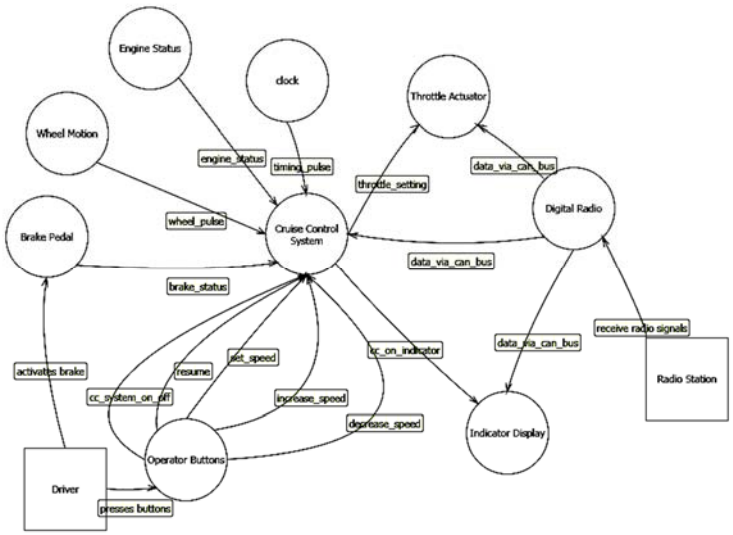

Figure 3: Data Flow Diagram of Self-Contained Cruise Control System

An automotive control system now includes dozens of embedded chips running millions of lines of code. The attack surface includes remote key systems, satellite radios, control units with wireless connectivity, Bluetooth connections, dashboard Internet links, and even wireless tire-pressure monitors. A DFD of a CC system now looks like the one shown in Figure 4. A digital radio with access to the CAN bus could potentially compromise other devices that use that bus, such as brake and throttle controllers. What is the risk that a compromised entertainment system could use that connectivity to control braking? Examples mentioned in Section 1 demonstrate that compromising a secondary component such as the entertainment system or dashboard displays too often has led to compromises of the driving controls. Threat modeling provides a way to identify and mitigate such risks.

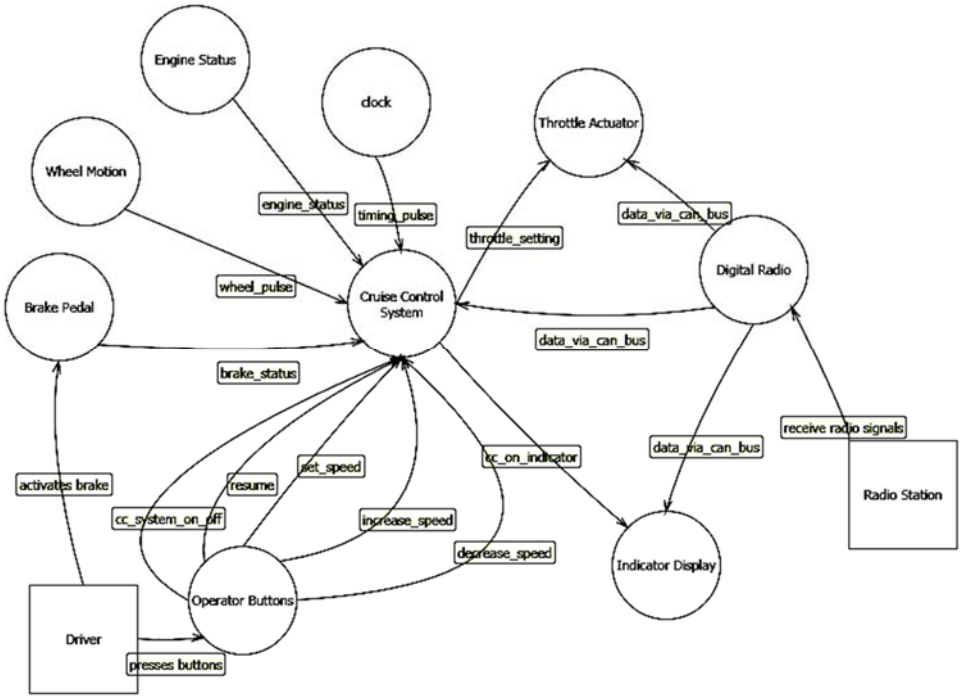

Figure 4: Data Flow Diagram of Cruise Control System

2.3 Architecture Description Language (ADL)

The entertainment system should not be able to apply the brakes. An essential part of a security design is to describe the roles, entities, and relationships that exist or should exist to perform certain actions. Most systems incorporate security services for authentication and access control to enforce those roles and relationships. The collection of such security services is often referred to as the *security architecture*.

Box and line drawings such as the DFD shown in Figure 4 informally represent what is called a *component and connector (C&C) view* of an architecture, which can also be described using an ADL such as AADL or a functional modeling language such as the Unified Modeling Language (UML). AADL lets us describe C&C types and information more precisely than we could with a DFD and supports analysis of the description. For example, an AADL description of the connection between a CC software controller and the throttle actuator specifies the type of data transferred, the direction of the transfer, the port on the CC controller that sends the data, and the port of the throttle actuator controller that receives the data. We can define well-formedness rules and analyze the described architecture using those rules to determine conformance, as we show in Section 5.

A designer can also define system-specific attributes for a C&C model. For example, the model could be annotated with the security risks identified by threat modeling and with engineering decisions made to control access and enforce any authorization requirements. As we show in later sections, an AADL model enables an analysis of a design before it is implemented, resulting in a specification that provides unambiguous guidance for developers.

2.4 Assurance

A well-structured design is not necessarily a secure one. The automobile attack examples described in Section 1 did not attempt to compromise any of the existing connections specified in Figure 3 on page 8. Instead, the attacks created a new connection. Add-ons such as an infotainment system extend the attack surface and provide an opportunity for a compromise. A security assessment must evaluate how well the engineering design decisions reduce the likelihood of an attack-created connection; for example, how compromises of third-party components are addressed.

The security problems arising with automotive systems share characteristics similar to safety issues the U.S. Federal Drug Administration (FDA) encountered with infusion pumps. A patient-controlled analgesia infusion pump is used to infuse a pain killer at a prescribed basal flow rate that can be augmented by the patient or clinician in response to patient need within safe limits. Infusion pumps, in general, have reduced medication errors and improved patient care by allowing for a greater level of control, accuracy, and precision in drug delivery than more labor-intensive techniques. From 2005 through 2009, 87 infusion pump recalls were conducted by firms to address identified safety problems observed across multiple manufacturers and pump types. The FDA became aware of many of the problems only after they occurred in the field.

One of the first steps in a safety engineering process is classifying the health risks for the pump (see Table 3).

Table 3: Infusion Pump Hazards and Health Risks

Hazards		Health Risks	
Software	Environmental	Overdose	Trauma
Operational	Mechanical	Air embolism	Exsanguination
Electrical	Hardware	Infection	Electric shock
Biological and chemical	Use	Allergic response	Underdose
All		Delay of therapy	

Specific hazards include air in the drug delivery line, tampering (e.g., by a patient during home use to adjust drug delivery), network errors, false alarms or the lack of an alarm caused by an out-of-calibration sensor, improperly set alarm priorities, incorrect settings of alarm thresholds, and software runtime errors. For serious hazards, an alarm should sound.

After an analysis of pump recalls and the occurrence of adverse events, the FDA concluded that many of the problems appeared to be related to deficiencies in device design and engineering [FDA 2010]. Those defects had not been found during development by testing and other methods. Such defects need to be identified as a design evolves, and a design review should confirm that faults associated with important risks (including attacks) have been identified and mitigated by specific design features.

Mitigating the spectrum of infusion pump hazards requires an integrated hardware and software solution. A pump's software must be designed to analyze the data from multiple sensors to correctly identify and respond to the identified adverse conditions. We need to evaluate the integrated hardware and software design.

Safety engineering has used an assurance case to show that systems satisfied their safety-critical properties [Kelly 1998, 2004]. For that use, they were called safety cases.

> ***Assurance case****: a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system's properties are adequately justified for a given application in a given environment* [Kelly 1998].[11]

An assurance case includes

- claims and subclaims for what we want to show
- arguments for why we believe each claim is met
- evidence such as test results, analysis results, and so forth that support each argument

An assurance case does not imply any kind of guarantee or certification. It is simply a way to document the rationale behind system design decisions.

Confirming the top-level claim that a pump is safe depends on showing that health risks have been mitigated. For example, an overdose could be caused by an environmental hazard such as a high temperature that leads to a pump malfunction. There is an internationally accepted standard for diagraming an assurance case,[12] and a graphical representation of an assurance case for an infusion pump appears in Figure 5 on page 14.

The use of an assurance case provides a formal structure for engineering design reviews. Such a review should consider if the argument and evidence provided by the developer might be insufficient to justify the claim. For example, does information exist that contradicts or rebuts a claim? The cause can be a combination of a poor argument and insufficient evidence. Are there specific conditions under which the claim is not necessarily true even though the premises (i.e., evidence) are true? For example, could an operational condition other than temperature, humidity, and air pressure lead to a pump malfunction? Is there sufficient evidence to justify the claim? The evidence supplied could have been provided by a simulation. How well does the simulated model represent projected operational conditions? Does the design mitigate sensor failures?

Not only can we specify an architecture using AADL, but extensions to the language enable an architect to construct an assurance case, add assurance attributes to the architecture, and describe the argument as rules to be analyzed and documented by an AADL tool (e.g., Resolute) [Gacek 2014].

---

[11] Assurance cases were originally used to show that systems satisfied their safety-critical properties. For this usage, they are called safety cases. The notation and approach used in this report have been used for over a decade in Europe to document why a system is sufficiently safe [Kelly 1998, 2004]. In this report, we extend the concept to cover system security claims.

[12] http://www.sei.cmu.edu/dependability/tools/assurancecase/

3 Analysis Framework for Security Assurance

Establishing security requirements depends on an understanding of how systems have been compromised, the resources and motivations of possible attackers, and the business consequences of a system compromise. The role of an AADL model is not necessarily to create security requirements but rather to help verify that an architectural design meets them. For example, as shown in Table 4, a data flow can be the target of tampering and information-disclosure threats. Threat modeling analysis could lead to a requirement that all connections capable of transporting a particular category of data should be encrypted. An analysis of the data flows in an AADL model could then verify that a design satisfies that requirement.

Table 4: Threats and Data Flow Elements

Element	Interactors	Data Flows	Data Stores	Software Processes
Spoofing	◊			◊
Information disclosure		◊	◊	◊
Tampering with data	◊			◊
Repudiation		◊	◊	◊
Denial of service		◊	◊	◊
Elevation of privilege				◊

AADL models can play a significant role in architectural tradeoff analysis. The success of a design is often determined by how well it satisfies a desired combination of architectural attributes such as usability, performance, reliability, and security. For example, an AADL model can be used to analyze the additional time required to encrypt a communications link to see if it violates latency requirements [Lewis 2009].

An essential part of a security design is to describe the roles, entities, and relationships that exist or should exist to perform a set of actions. Most systems incorporate security services for authentication and access control to enforce those roles and relationships. The collection of such security services is often referred to as the security architecture. Access control and authentication requirements can be verified using an AADL.

3.1 Resolute: An AADL Analysis Tool

Research funded by the Defense Advanced Research Projects Agency's (DARPA's) High Assurance Cyber Military Systems (HACMS) developed an extension of AADL called Resolute that describes the assurance rules a system should satisfy to justify a claim [Gacek 2014]. For example, a design rule could specify that user input is always verified.

The syntax of Resolute is inspired by logic programming. Each rule defines the meaning and evidence for a claim. The meaning of a claim is given by a text string in the rule that is parameterized by the claim's arguments. The body of the rule consists of an expression that describes sufficient evidence to satisfy that claim. Claims can be parameterized by AADL types (e.g., threads, systems, memories, connections), integers, strings, Booleans, or sets. The following are examples of Resolute rules:

```
only_receive_decrypt(x : component) <=
 ** "The component " x " only receives messages that pass Decrypt" **
 forall (c : connection).
    (parent(destination(c)) = x) =>
     is_sensor_data(c) or only_receive_decrypt_connection(c)
only_receive_decrypt_connection(c : connection) <=
 ** "The connection " c " only carries messages that pass Decrypt" **
 let src : component = parent(source(c));
 unalterable_connection(c) and (is_decrypt(src) or
 only_receive_decrypt(src))
```

An assurance case is initiated in Resolute by adding a Prove statement for an AADL component. A Prove statement consists of a claim applied to some component such as the following example for the claim that the Motor Controller thread only receives input from the ground station:

```
prove only_receive_ground_station (MC)
```

To evaluate a Prove command, Resolute acts like a theorem prover but with assurance-case rules replacing logical rules. A successful proof produces an assurance case. Resolute can demonstrate that the rules representing the design decisions have been appropriately applied but cannot justify the choice of rules. For example, when the risk of a SQL injection vulnerability exists because user input is used to create database queries, invalid input can enable an attacker to modify the contents of a database. Assume that assurance for eliminating the risk of a SQL injection is based on the rule that user input is verified. A Prove command for that rule succeeds if user input has been checked at some point along a data flow. But verification is not validation that the application of the rule reduces the risk of a SQL injection. Input verifications are difficult to validate for SQL injections, and the CWE recommends alternative mitigations that have a higher level of assurance.

3.1.1 Developing Assurance Cases for Security

Security and safety engineering identify possible risks in significantly different ways. Using the Goal Structuring Notation (GSN) shown in Figure 5, a safety engineer could develop the assurance case argument based on the hazard categories listed in column 1 of Table 3 (on page 10), as shown in Figure 6.

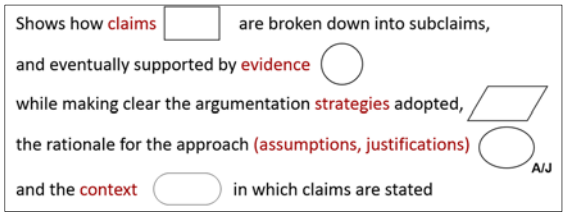

Figure 5: Goal Structuring Notation [Kelly 2004]

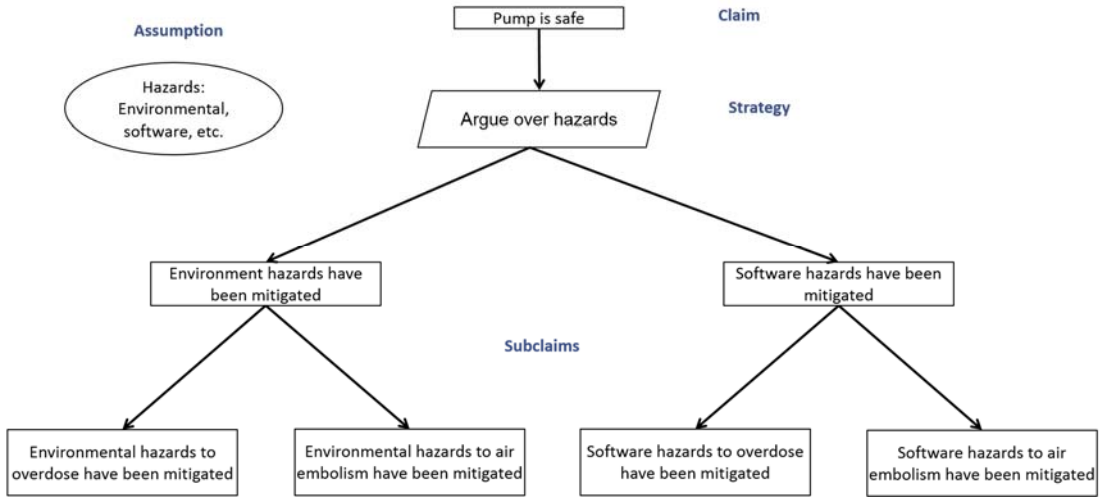

Figure 6: Assurance Case Argument

The health risks within a category could change for this generation of pumps, but the categories are not likely to change. The security-threat categories do change because of the adversarial aspects of security. For example, in the 90s, attacks typically targeted network or operating system weaknesses. As better engineering reduced the effectiveness of such exploits, attack tactics shifted to finding exploitable defects in the new target category of application software [Nagaraju 2013]. An email sent to RSA employees included an Excel attachment with malware that exploited a vulnerability in Adobe Flash. The compromise enabled the attacker to take remote control of an RSA employee's computer and use it as a backdoor into the RSA network.[13] The CWE documents over 900 such weaknesses.[14]

Too often, the architect for a CPS creates a design defect that is well-known to an IT architect. For example in July 2010, malware called Stuxnet targeted specialized industrial control equipment made by Siemens [Mills 2010, McGraw 2010]. This malware enabled the attacker to modify how the control system managed a physical system, such as one for water treatment. The operating system is typically the case-loaded system libraries at runtime. Stuxnet exploited a defect in

[13] http://www.cnet.com/news/attack-on-rsa-used-zero-day-flash-exploit-in-excel/

[14] http://cwe.mitre.org

the implementation of that feature to load attacker-developed libraries—a defect that had been eliminated from IT operating systems for over 10 years.

An objective of applying software assurance techniques is to increase our level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner.[15]

The justification for a level of confidence can be documented in a security assurance case. Architectural security models and the use of tools such as Resolute to verify such models can contribute to that justification. The next section explores how such models and tools could support security analysis.

[15] https://en.wikipedia.org/wiki/Software_assurance#cite_note-1.

4 STRIDE Analysis of the Infotainment System

As described in the prior section, security requirements must first be established, based on the threats considered to be important to the system. Threat modeling is used to determine the important security concerns that must be translated into design considerations and incorporated into the model. The architect must systematically determine the potential consequences of each threat. To demonstrate how this process would work, we applied the STRIDE threat modeling approach to a car infotainment system.

The name STRIDE [Hernan 2006] is an acronym based on the initials of the six threat categories: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. These categories are not mutually exclusive, and complex attacks may involve a combination of them. However, they provide a useful set that non-security experts can use to reason about security threats.

This technique is a way to walk through the major categories of threats a system may face and to systematically determine the potential consequences of each threat category, as that type of threat applies to the particular system under scrutiny. Below, we consider the security implications of each of the six STRIDE threat categories on the automotive electronics system presented in Figure 4. For each category, we describe

1. how the threat might be realized: the conditions under which it might occur
2. the implications of this threat on the infotainment system design and, in particular, how the system design is represented in an AADL model
3. the risks that arose from a consideration of the threat in the infotainment context
4. questions about the architecture of the infotainment system and its implementation that will inform the security analysis
5. the security property that is violated if this threat is not mitigated

Mitigating many of the STRIDE threats involves establishing trust boundaries. A trust boundary can be thought of as a line drawn through a program. On one side of the line, data or an agent (a user) is untrusted. On the other side of the line, the data or agent is assumed to be trustworthy. For example, input validation could be required for data to cross a trust boundary. But data may only be trusted from specific sources, and in those instances a trust boundary could have authentication and authorization requirements.

In the following six subsections, we use the five categories above to describe each STRIDE threat as it applies to the infotainment system.

4.1 Spoofing

Spoofing is an attack in which people (or programs) represent themselves as something other than what they truly are, with the intent of gaining authorized access to resources for which they should be unauthorized. A successful spoofing attack is one that allows an attacker to foil or avoid authentication.

4.1.1 Conditions Under Which Spoofing Might Occur

Spoofing can occur when the source or destination of a message is not properly trusted (e.g., via authentication), but the requested action in the message is still performed. Spoofing can be successful if the attacking component can steal another component's identity to appear authentic or if other components do not demand proof of authentication.

4.1.2 AADL Implications of Spoofing

AADL can define a system scope, so we can check whether information flows across the (car) system boundary. To do that, we need to classify components inside and outside the system and determine that each has been identified and has an associated credential, such as a certificate. In the architecture, we can establish a placeholder, abstract data type for authentication that is passed between components. This data type could be modeled as a pattern in AADL (with an authenticator, an error-handling component, and entities that need to be validated).

4.1.3 Spoofing Risks

When considering spoofing attacks, we must think about these general design weaknesses that would allow spoofing to occur:

1. There is no authentication, or the authentication mechanism has been broken or bypassed.
2. An external (third-party) component is mistakenly trusted as authenticated.

We can test for these weaknesses by analyzing an AADL model.

4.1.4 Questions About Spoofing

From an analysis perspective, we must answer the following questions:

1. Does a service exist in the system that provides proof of the integrity and origin of data?
2. Have trust boundaries been established and documented appropriately?
3. What connections and interfaces does the system have to the outside world?

The answers to these questions must be modeled in AADL to establish how the system will address the risk of spoofing.

4.1.5 Security Property Violated by Spoofing

Spoofing attacks violate the Authentication property.

4.2 Tampering

The objective of tampering is to perform unauthorized modifications to data or services. In a tampering attack, an attacker makes a modification to change the system's runtime behavior through unauthorized access to the data or service.

4.2.1 Conditions Under Which Tampering Might Occur

Tampering could occur if the infotainment system contains data stores, as it almost certainly will. If the data's encryption is strong, the attacking component can attempt to simply corrupt the data,

rendering the system unusable or less usable. If the encryption is weak (or non-existent), the attacking component can modify the data, perhaps without being detected (particularly if the system does not log messages and transactions). This risk can also affect data in motion, if it is not adequately encrypted.

4.2.2 AADL Implications of Tampering

In AADL, a component's encryption property (such as a data store) can be specified. Similarly, an encryption property can be associated with any connection (the data-sharing mechanism between two components). Encryption can be specified over multiple AADL components, and an encryption property can be specified on the data itself. The communication mechanism that accesses the data may stipulate an encryption mechanism. If multiple logical data channels are modeled, you can specify for each channel a set of properties that include, among other things, their encryption properties. This group of logical channels is called a *feature group*.

4.2.3 Tampering Risks

When considering tampering attacks, we must think about these risks:

1. If tampering is not detected, any assumptions about system behavior are invalidated.
2. Trust boundaries will affect how components are grouped and hence where encryption will and will not be applied.
3. Trust boundaries will affect when and where an actor is authenticated or re-authenticated.

Therefore, trust boundaries must be explicitly specified in any AADL model.

4.2.4 Questions About Tampering

From an analysis perspective, we must answer the following questions:

1. Have the critical pieces of data been defined?
2. Is the encryption of the data sufficiently strong (given the criticality of the data)?
3. Is a system-wide encryption scheme uniformly applied?
4. How easily can the specified encryption be broken?
5. Have authentication and access control mechanisms been planned?

The answers to these questions must be modeled in the AADL model.

4.2.5 Security Property Violated by Tampering

Tampering attacks violate the Integrity property.

4.3 Repudiation

Repudiation can occur when a system does not properly track and log the actions or changes of users (or other system actors). In such a case, malicious users may be able to forge an identity so that their actions (attacks) will be difficult to trace and might go undetected for an extended period of time.

4.3.1 Conditions Under Which Repudiation Might Occur

Repudiation could occur if an attacking component changes data (e.g., state information, driver information) without the possibility of that change being traced back to that component. Such changes will be difficult to detect and prevent if, for example, the components store data without using a data hash.

4.3.2 AADL Implications of Repudiation

A typical pattern used to guard against repudiation attacks would require a logging function in the architecture. The elements to be logged must be determined (typically from the data architecture), and the frequency and limits for logged data must be specified, since the logging actions might impact performance. To support non-repudiation, the design should clearly designate the existence of a logging function and the ways in which the data to be logged are communicated and stored by this function. Mechanisms that authenticate those with legitimate access to logs must be established. The AADL model can then be checked to ensure that the logging function is employed correctly and ubiquitously.

4.3.3 Repudiation Risks

When considering repudiation attacks, we must think about these risks:

1. If no logging is being done, repudiation cannot be ensured.
2. If authentication is inadequate, logging is meaningless.

4.3.4 Questions About Repudiation

From an analysis perspective, we must answer the following questions:

1. Which parts of the system are trusted?
2. How is trust established with third-party components?
3. How is authentication managed? Can a component's identity be spoofed? For example, does a system-wide authentication service exist that can be asserted to be genuine with high assurance?
4. Does a service exist that provides proof of the integrity and origin of data?
5. Is transaction data logged?

4.3.5 Security Property Violated by Repudiation

Repudiation attacks violate the Non-Repudiation property.

4.4 Information Disclosure

Information disclosure enables an attacker to gain (potentially sensitive) information about a system, possibly leading to a data leak, a privacy breach, or the disclosure of information that could be used to launch additional attacks.

4.4.1 Conditions Under Which Information Disclosure Might Occur

If an attacker can read a process's state, capture information in transit, or break into a system's database, sensitive information might be disclosed. For example, if the system uses the broadcast of

messages or publish/subscribe features (where subscriptions are not managed via authentication and authorization), an attacking component might be able to use this design weakness to steal information in transit. Similarly, if an attacking component can spoof a recipient's identity, it could collect (and steal) poorly protected information intended for the legitimate recipient.

4.4.2 AADL Implications of Information Disclosure

From an architectural perspective, to address information disclosure risks, we need to design into the system the same authorization and authentication safeguards that also address spoofing, elevation of privilege, and tampering risks.

4.4.3 Information Disclosure Risks

When considering attacks that could lead to information disclosure, we must think about these risks:

1. Disclosure of operational information can lead to other security or availability problems.
2. Disclosure of customer or user information can lead to a loss of reputation, as well as increased likelihood of other kinds of attacks.

4.4.4 Questions About Information Disclosure

From an analysis perspective, we must answer the following questions:

1. What data in the system is operational, affecting the state of the system?
2. How is information transmitted around the system (what mechanisms, what properties)?
3. Where is information about the communication medium (e.g., publish/subscribe) and its properties documented?

4.4.5 Security Property Violated by Information Disclosure

Information disclosure attacks violate the Confidentiality property.

4.5 Denial of Service

A denial of service (DoS) attack is an attempt to make a computational or network resource unavailable to its intended (legitimate) users. This attack is typically accomplished by flooding the system with useless traffic or service requests.

4.5.1 Conditions Under Which DoS Might Occur

DoS attacks, which are perhaps the most critical security risk for an automotive system, can affect safety-critical performance and availability properties. An attacking component, which may or may not be authorized in the system, could attempt to saturate the available system channels with communication requests.

4.5.2 AADL Implications of DoS

The desire for performance modeling, now a well-established capability of AADL, was one of the driving factors that led to AADL's creation. To detect and respond to a DoS attack, the system would need to include monitoring and control functionality that monitors network traffic and responds to unacceptable anomalies by shedding load or changing configurations. Employing

AADL allows limits to be associated with the performance properties of (previously identified) critical portions of the system, and these limits can be monitored by the monitoring component. Furthermore, the consequences and probabilities of a component failure due to a DoS attack need to be assessed so that mitigations can be planned, such as increasing the cost for the attacker. Potential DoS attack vectors must be identified through a distinct threat modeling activity to establish where monitoring activities should occur.

4.5.3 DoS Risks

A successful DoS attack can affect the system's safety-critical properties. For example, if the brake system is under attack, it may not be able to respond to legitimate commands in a timely fashion, and that delay could affect a car's safety by making the vehicle unresponsive to the driver's actions.

4.5.4 Questions About DoS

From an analysis perspective, we must answer the following questions:

1. What is the minimum (safe) operating state of the system that can be guaranteed despite failures?
2. What system component is receiving external packets, making it a potential target for DoS?
3. How much "headroom" is in the processing capacity of the computers or network?
4. How are "bad" components shut down or bypassed?
5. Does a central communication "manager" exist? If so, does it have the ability to inspect (and scrub) communications, limit rates, or monitor system network traffic to detect and respond to anomalies?

4.5.5 Security Property Violated by DoS

DoS attacks violate the Availability property.

4.6 Elevation of Privilege (EoP)

An EoP attack occurs when an attacker obtains authorization permissions beyond those initially granted, typically by exploiting a weakness—a programming error or design flaw—in the system. As a result of this exploit, the attacker can perform unauthorized actions.

4.6.1 Conditions Under Which EoP Might Occur

EoP involves an attacking component gaining access to data or resources beyond what its permissions allow (in terms of its group membership and data access rights [read/write/execute]). This unauthorized access can occur due to "stealing" the identity of another component (via spoofing or information disclosure) or due to a jailbreak-type attack.

4.6.2 AADL Implications of EoP

An AADL model can define data types with allowable enumerated values and then check the compatibility of data shared between components such that they comply with the data access rules on those types. We have defined two such properties: group membership and access mode.

4.6.3 EoP Risks

Elevation invalidates all the other security properties and mechanisms built into the system. An EoP attack may facilitate any of the other attacks described in this chapter.

4.6.4 Questions About EoP

From an analysis perspective, we must answer the following questions:

1. Does the system employ any form of data execution prevention?
2. Do multiple levels of privilege exist? If so, does the system execute processes with least privilege?
3. Are system components "certified"? Can the validity of that certificate be confidently checked so a compromised component can be identified?
4. Does the system employ anti-virus software to reduce the likelihood of component attacks from known malicious code? Is that software kept current?
5. Does a patching mechanism exist? If so, are patches automatically (and promptly) applied?
6. Has any diversity been designed into the system such that an attack on one part does not permit compromise of the entire system?

4.6.5 Security Property Violated by EoP

EoP attacks violate the Authorization property.

4.7 Using the STRIDE Model

These STRIDE threat categories provide an analysis framework of possible threats against a system. To analyze a specific system, we use the framework to characterize and analyze specific threats that are instances of one or more STRIDE categories and to construct a system-specific threat model. In Section 5, we show how we instantiated two specific threat scenarios for the car infotainment system—one for EoP and one for confidentiality—and modeled the mitigations of such threats in AADL.

Over the course of this project, we were interested in exploring other ways to improve threat modeling efforts, especially among those who may be relatively new to the process and have limited security expertise. To that end, we assembled two groups of people to play the Elevation of Privilege card game from Microsoft, which is designed to facilitate discussions of system threats among individuals with and without security experience. Playing that game was a useful way to seed security discussions about a system even with minimal information about it; for example, at the point when the architecture is still being developed and no software has been written or even fully specified yet. We believe that threat modeling in general and the EoP game in particular are complementary to early phases of the system architecture and design lifecycle. We describe the results further in Appendix A.

5 Example Solution Approach for Security Threats

This chapter provides a brief introduction to AADL and explores some of the ways in which the language supports the security design structures needed for the automotive infotainment example. The detailed steps for building an AADL model are described, along with how to represent and check the selected security requirements.

5.1 AADL Description

AADL and its extensions support

- architecture specification and validation
- specification of functional components, their interfaces, and their interactions
- mechanisms for analysis representation (i.e., qualitative analysis and rule checking of system specifications that define properties, mathematical evaluation, and model checking)
- verification of approaches and outcomes (e.g., Mathematically verify that a quantitative requirement is met by the architecture, and ensure that a claim about components, interactions, and associated properties is upheld throughout the architecture.)

AADL specifies a static representation of the system architecture by the use of component types that can model software functionality, software runtime specifications, execution hardware, hardware and protocols used for connections, and related components such as sensors and actuators. The language supports the notation for logical flows, binding of software to execution hardware, and modal operation of both software and hardware. The modeling notation is structured to support a number of consistency checks to be developed in a modeling tool set.

The language defines the following component types:

- system
- device
- thread and thread group
- process
- processor
- bus
- memory
- subprogram

These components represent abstractions of their named entities and interface with each other through ports (and port groups) features. A system is modeled, in general, by connecting component types within a system implementation. AADL allows for modeling an architecture by separating concerns and having multiple views of the architecture (e.g., logical data/control view; execution platform view; and process and thread view).

In the language, *properties* capture the essential characteristics of components and allow for component constraint specifications, documentation of expected (to be implemented) capabilities, execution specifications, latency values, and so forth. For example, a software component that performs encryption can have a property that specifies the execution time latency (e.g., 3 ms) and another property that specifies the type of encryption the component is to perform (e.g., Rivest-Shamir-Adleman [RSA][16]). If needed for the particular modeling effort, additional detailed properties relevant to the encryption type could be specified (e.g., data [word] size).

Properties can be defined by an AADL user to extend the core language. Properties capture important design constraints and architectural specifications that support the mathematical analysis of the overall architecture. For example, the latency property specified for a number of components can support analysis for computing the overall end-to-end latency of a flow. Properties can also capture boundaries on system resource requirements such as memory footprint (in words), power consumption, and scheduling protocols.

5.2 Security Analysis for the Infotainment Example in AADL

Securing a computer system involves establishing and enforcing security policies such as a requirement that all users must be authenticated. The design of an architecture includes designating Policy Enforcement Points (PEPs) such as where authentication is enforced in a data flow. An essential security policy is that untrusted input should not be used when constructing commands that will be executed by other components such as a database query (SQL injection) or in JavaScript programming instructions that will be executed by an external user's browser (Cross-Script Injection). For a database query (SQL injection vulnerability) as shown in Figure 7, the data is untrusted as supplied by the user but must be trusted when it reaches the database server.

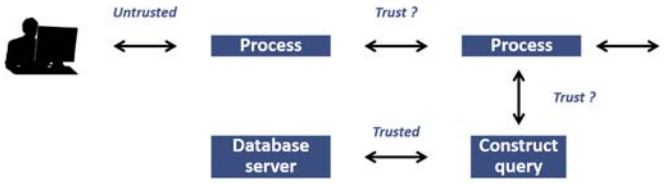

Figure 7: Trust

Security analysis starts with a description of the attack surface. As described in Section 2, it includes communication channels and access rights. Over time, the automotive attack surface has grown from the initial OBD port to now include remote key systems, satellite radios, telematics control units, Bluetooth connections, dashboard Internet links and even wireless tire-pressure monitors. But the expanded attack surface has not been matched by the addition of security functions to monitor and control access. The threat modeling framework, described in Section 4, draws on knowledge of successful attacks to analyze how a proposed design could be exploited and analyzed ways that a design or architecture could eliminate or at least reduce such risks.

[16] For the definition of RSA, go to http://searchsecurity.techtarget.com/definition/RSA.

Invalid input of some form is the root cause of many compromises. For many of the automotive ones, the format of the input (e.g., input to the braking system) was valid, but the source (e.g., the infotainment system) was invalid.

There were no access controls on the infotainment system's access to the CAN bus. The architecture for the automotive control system should include a trust boundary as shown in Figure 8. A trust boundary defines a segment of an architecture with a requirement that, within the boundaries, access is controlled. Examples discussed in this chapter show how to specify a trust boundary using AADL extensions.

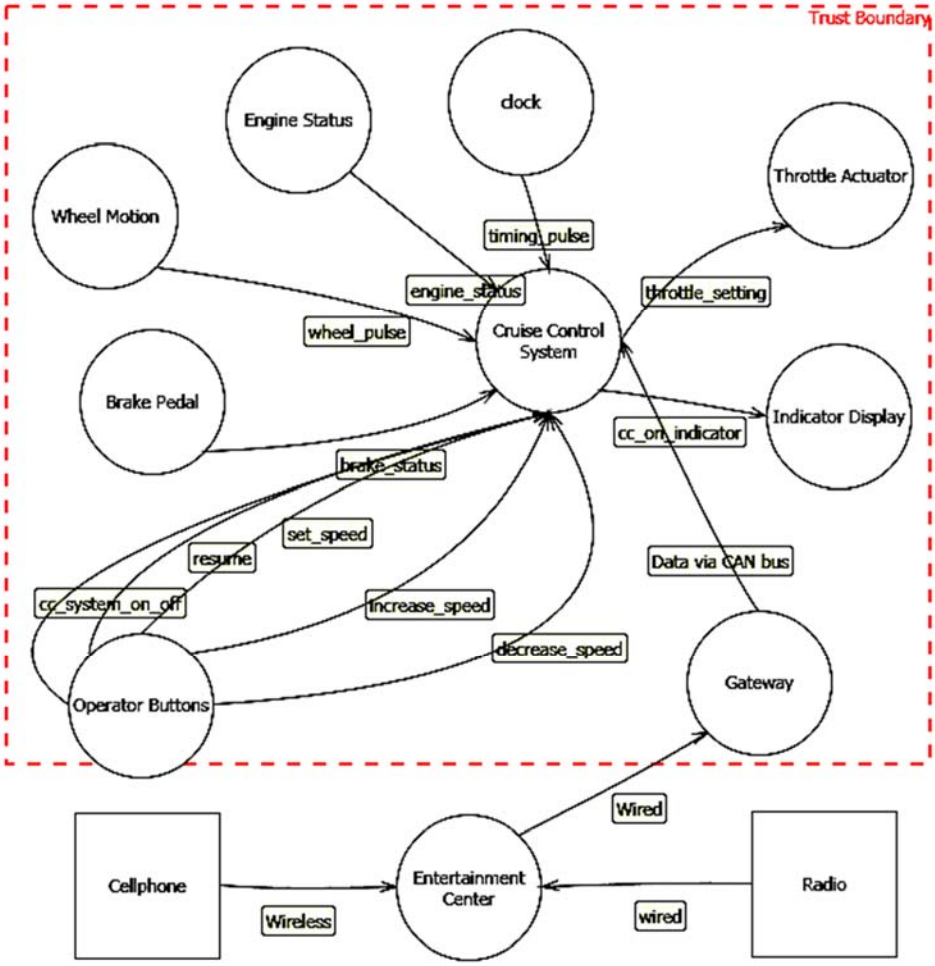

Figure 8: Trust Boundary for CAN Bus

There are several ways to design such a trust boundary for an automotive electronics system. In what follows, we focus on the portion of the automotive electronics system related to CC, along with the interactions between the CC system and other automotive electronics components.

Trust could be assumed among the components of a CC system, since they are provided by the manufacturer and are not runtime replaceable. Because we assume that there are no security risks associated with a network consisting of only CC components, one mitigation tactic is to create a sub-network consisting of only those components. We can create such a network by having the

gateway component in Figure 8 act as a PEP to block all access to the sub-network from non-CC components. This gateway could implement a second sub-network consisting of the infotainment system and the display. Such an approach would be sufficiently secure as long as the gateway controls all connectivity to the CAN bus. The security problems are more difficult if the CAN bus can be accessed wirelessly. For example, a DoS targeting the wireless access point could be used to compromise the control system. Should such access be allowed if that risk cannot be sufficiently mitigated?

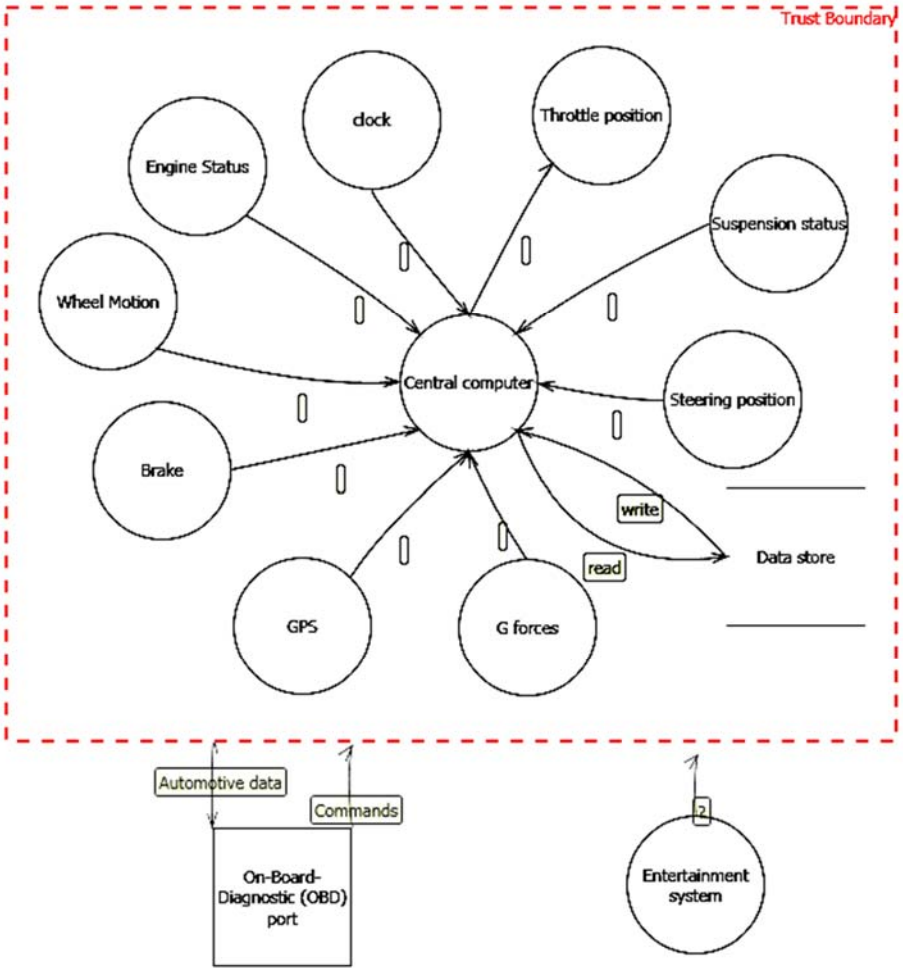

Figure 9: Monitoring

A CC system is just one of the electronic control systems (ECSs) that can be installed. Others include electronic stability controls, driving monitoring, and maintenance and diagnostic packages. For example, an ECS could include a steering-wheel-angle sensor to determine the driver's intended rotation, a yaw rate sensor to measure the rotation rate of the car, a lateral acceleration sensor, a wheel-speed sensor, and a longitudinal acceleration sensor that might provide additional information about road pitch and a second source of vehicle acceleration and speed.

The architectural issues for security can arise with the use of third-party OBD-port plug-ins with real-time access to sensor data or with remote access to stored data such as that used for maintenance. For example, after an accident occurs, the saved record of automobile behavior could be analyzed to determine whether the cause was driver or vehicle failure. The system should be designed so that a driver cannot use the OBD port to clear that data store. AADL can be used to model a trust boundary between the automotive control system and devices connected to the OBD port, and between the control system and other automotive components such as the infotainment system as shown in Figure 9.

5.3 Architectural Models in the Design Process

Architectural models can be developed in several ways. In this research example, we performed the model development process in three phases: (1) Focus, (2) Build, and (3) Analyze, each explained below.

5.3.1 Focus Phase

The goal of the Focus phase is to create a foundation for analysis by identifying elements of the model problem and conceptualizing the functionality of the key quality attributes and hardware and software components. For our model problem, this exercise is based on understanding security aspects of the automotive ECS and how they could be captured and evaluated. Performing this activity on a real project would identify a broader set of key quality attributes (e.g., requirements for performance, capacity, communication [bandwidth], safety, reliability). These quality attribute requirements would form the basis against which the architectural model could be analyzed, quantified, and evaluated.

We applied the STRIDE framework to develop a threat model for the system and identify the critical security mechanisms needed to address the security concerns. In our automobile example, many of the security mechanisms rely on authentication, which led us to add an authentication service, the associated authentication server hardware, and logical communication channels that supported the authentication service and protocol (and that would eventually be mapped to hardware realizations).

Also in this phase, we developed usage scenarios for exhibiting the operational and logical behavior that would allow us to evaluate how well the architecture met the STRIDE categories. We discuss two of the usage scenarios in detail later in this chapter.

5.3.2 Build Phase

The Build phase was decomposed into three parts: (1) building the initial model, (2) identifying gaps when comparing that model to the security concerns developed from the STRIDE framework, and (3) formulating the logic rules to be used in the Resolute model checker.

The activities in building the initial model for this investigation were the same as for any modeling effort. We initially developed a context model for the model problem: a vehicle control system with major functional components that included

- the CC system
- an infotainment system

- stability control
- the sensors that provided data on the physical vehicle and its environment to the control systems
- the associated actuators controlled by the systems
- system data to be displayed on the infotainment system
- computing hardware
- communication physical layers such as Ethernet, CAN bus, and radio (e.g., Bluetooth)

We chose AADL model components based on the degree of abstraction necessary to capture functionality. For example, sensors were modeled as simple devices because we did not need to model the sensors' execution characteristics. We developed two views of the system: a logical view and a deployment view[17] to support our analysis goals. We focused on identifying application patterns and communications that are typical in real-time control and CPSs to create a model that supports analyzing how the architecture would respond to security requirements.

As we built the model, we realized that crucial elements needed to address security concerns were not part of our initial conceptualization, because they were not driven by the primary domain concerns of managing the car's sensors, actuators, driver controls, and displays. For example, we decided to add authentication functionality to the architecture in the form of a software service that performed authentication, the associated execution platform for this service, and logical connections between the authentication service and components that needed authentication. These components were added into their respective views.

Models for analysis are built with an eye towards validating system requirements. To that end, properties are added to the components to specify a constraint (reflected in the requirements) or as a placeholder for data that can only be determined when the software is actually executed. In using the threat scenarios developed from the STRIDE model to generate security requirements, we identified properties that were not part of the AADL default property set. To address those properties, we needed to develop the security-specific properties described in Section 5.5 and determine how best to analyze them.

In some cases, the identified properties cannot be analyzed at the architectural level but serve as a constraint for downstream design and implementation, or serve as a placeholder for a property value that can be obtained by simulation. An example of the former is specifying RSA as an encryption property defining the algorithm that should be implemented. An example of the latter would be specifying an encryption protocol with a latency requirement that could be simulated in a communication simulation environment such as OpNet. The values obtained from the simulation could then be assigned back to the appropriate properties.

Resolute, the AADL tool described in Section 3, can be used to verify a model. Formulation of the Resolute rules involve expressing the model components particular to a claim in association with the logical relationship between or among components. For example, one rule, expressed in English, is that any component reading data marked with a confidentiality level of 1 should also

[17] For details on these two views, refer to the AADL reference book [Feiler 2012] or to a more general discussion of architecture documentation [Clements 2010].

have a confidentiality level of at least 1. This rule essentially ensures that any component handling data has the same or greater level of confidentiality as the data. The specifics regarding rule creation are described in the scenario descriptions later in this chapter.

5.3.3 Analysis Phase

The activities in the Analysis phase included ensuring the model had enough detail to meet validation requirements and using Resolute to determine if the defined security rules were satisfied. The example models were reviewed against the scenarios formulated in the Focus phase to ensure they were constructed correctly. Next, we instantiated the models and ran the Resolute rule checker. The results of Resolute are that either the rules were satisfied or they failed. In either case, but particularly if the rule failed, we checked the rule manually for correctness. Normally, various what-if analyses would be done in this phase to identify gaps in the application of the rule; for example, if a component was annotated with an invalid property. Once correctness is established, the rules could be applied to larger models.

Changing the architecture or specifying other quality attribute properties of a system can often uncover related but hidden dependencies that may result in the architecture not meeting the design requirements. For example, consider the case where a requirement to increase confidentiality is revised, necessitating a change in the encryption policy. Modifying the frequency of key changes along with the key size will increase the message size, and that, in turn, may increase bandwidth usage and power consumption. Increased computational complexity may increase the worst-case execution time, CPU usage, and power consumption. Running the analysis to check for the correct confidentiality level on data and software applications may indicate that the requirement is met, but running subsequent analysis for end-to-end latency, bus bandwidth usage, and power consumption may show that one or more of the requirements will not be met. Having one architectural model that can capture the properties and their interdependencies helps to discover early in the design lifecycle some of the interdependency problems that typically surface during integration. At that stage, those problems are typically expensive to discover and correct.

5.4 Model Problem and Associated Architecture

We created both a logical view of a plausible (but not exhaustive) set of application components (Figure 10) and a physical view of the execution platform. These views map each software component to a hardware component (Figure 11).

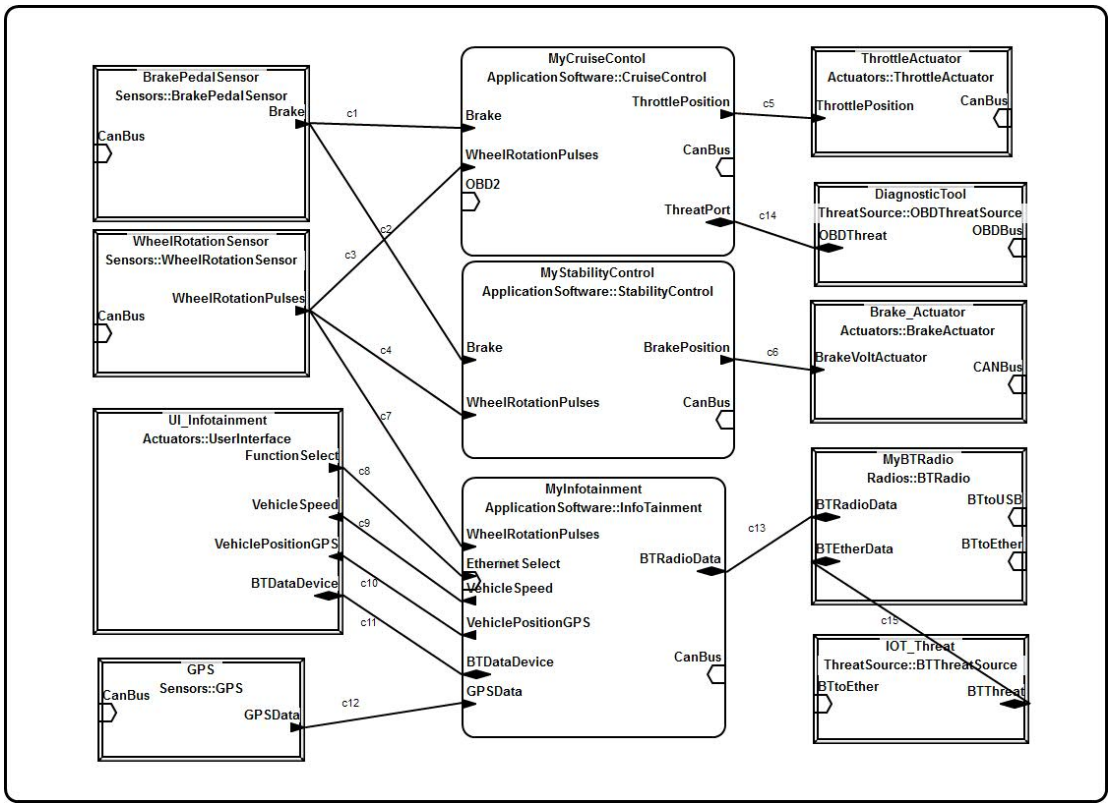

Figure 10: AADL Model Logical View of Automotive Electronics

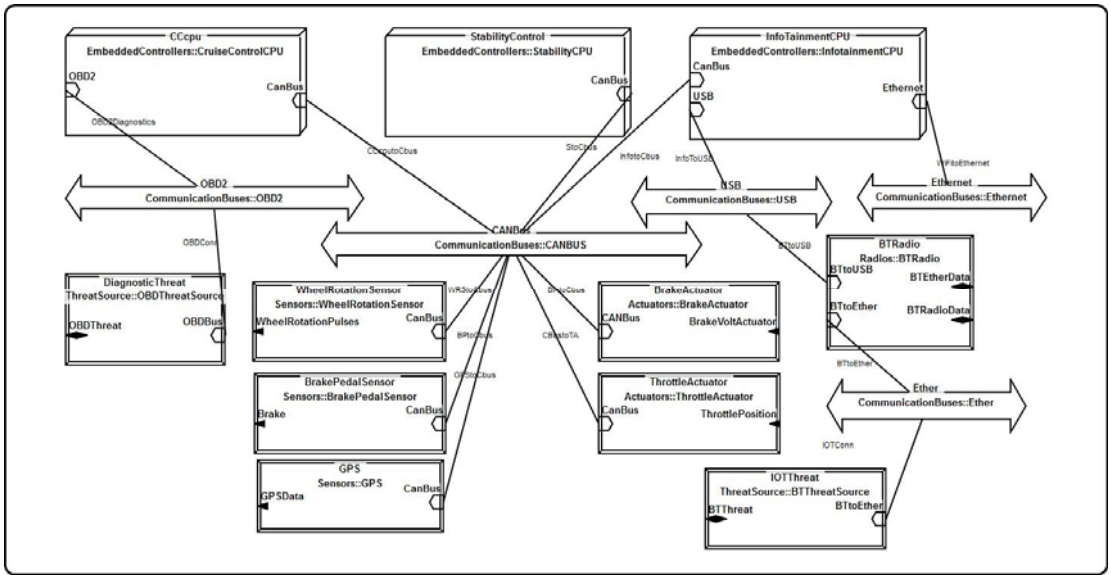

Figure 11: AADL Model Physical View of Automotive Electronics

Figure 4 on page 9 is a DFD of a CC system. Many of the components shown in that diagram can be mapped directly to either the system or device components as shown in Figure 10 and Figure 11 above (e.g., Wheel Motion Sensor, Brake Pedal Sensor). In some cases, abstractions are used to capture the essence of the objects and their interfaces. For example, the Operator Buttons in

Figure 4 are part of the automotive ECS and were represented in the Infotainment User Interface that we modeled in AADL. We paid particular attention to the Function Select port that provides data as to what function the user wants (e.g., vehicle data, navigation/GPS) and the data associated with the function. We aggregated the individual signals (data) coming out of the Operator Buttons element into a single interface (data port) in this model. A general guideline in architectural modeling is to represent enough detail to capture the essence of the situation being studied to perform analysis. If the signals (data) going from one component to another are conceptually similar and have no significant quality attribute differences (e.g., different latency goals), those signals can be combined into a single data element for modeling and analysis purposes and then be shown as a single connection to another component in the architecture diagram.

Figure 10 represents a component and connector view of our model. The intent of this diagram is to determine the interfaces, the data associated with the interfaces, and the logical interaction among components. Most of the threats associated with the STRIDE categories are based on some entity gaining access to the system via existing hardware interfaces, and the malicious activities are largely centered on accessing and manipulating data. Many of the deterrents to attacks involve detection of data manipulation or data protection via access protection or encryption. Hence the component and connector view is a key to reasoning about, representing, and analyzing the architecture.

We found it necessary to add components in our model to capture the notion of Internet devices, diagnostic devices (a diagnostic instrument via the vehicle's OBD interface), or external threats. These devices are not permanently connected to the vehicle but need to be represented in a static system view to provide a source of threats and an interface through which they interact with the rest of the system. AADL is a modeling and specification language, not a simulation language, so the existence of objects dynamically joining the system must be represented via static objects. We can then analyze the physical, logical, and data exchanges among such objects.

Figure 11 represents a physical view of the model problem. This view shows the processors, buses, and connection to the buses that represent the computing platforms and connection topology for the components of our model problem. For example, CruiseControlCPU contains connections to the CAN bus and the OBD connector (OBD2) used by technicians to read and write diagnostic information for the vehicle. Wireless devices connect to the system via radio links. Both communication mediums can be represented by the AADL Ether bus component that represents the air and the corresponding Bluetooth radio. The Ether bus represents the medium through which any Internet of Things (IoT) device such as a smartphone or iPod would attempt a connection. As part of any architectural design and for review and analysis, it is important to capture interactions at this level to ensure that the necessary protocols are in place. For example, a data encryption technique that is mapped to a software application must also be mapped onto a secure communication protocol as part of the (hardware) bus model specification.

The physical view of the system is not technically a deployment view, but in this example, the application components were bound to the processor using the AADL binding properties and a simple similar-name binding approach. For example, the CruiseControl system component was bound to the CruiseControlCPU processor, and the StabilityControl system was bound to the StabilityCPU processor, essentially making the physical view a deployment view. To keep the diagram uncluttered, this binding is not shown in Figure 11.

The logical and physical views (which are two views of the same system) allow designers and analysts to focus on security concerns specific to that view. Looking at a physical connection/communication view of the system reveals the paths by which hardware components interact. Note that security design decisions can affect other system quality attributes such as performance, reliability, and so forth. From a holistic system perspective, it is important to qualitatively understand the implications of software and hardware security design decisions with respect to other system quality attributes. For example, changing an encryption technique may adversely affect bus bandwidth. Using AADL to model various perspectives from a single model allows concurrent analysis of other quality attributes to ensure that the overall system requirements are being met.

5.5 Analysis Options for Our Example

Having developed an understanding of the threat types and associated semantics for the infotainment example, we needed to determine the best approach to both capture and analyze the security characteristics to ensure consistency of the security characteristics across a system model.

For some attack types, certain new architectural functions, possibly packaged as independent components, had to be specified. For example, we needed a software component that would function as an authentication server. Secondly, the external characteristics we needed to consider the security properties would have to be represented as AADL user-defined properties.

Designating the architectural components that provide authentication services would have to be done manually, by a person performing an architectural review. Checking for correct property-to-component associations is handled by the syntax of the property definition language in AADL. For example, when a property is defined, a list of the AADL architectural elements to which it applies can be specified. Checking for matching properties across components, across the system, or over the component hierarchy is more difficult, but modeling checking tools can be used.

For example, a component whose output has a confidential level of high can only communicate with components that can accept that level of confidentiality. We used Resolute to verify such claims. Next, we describe the syntax and claim formulation using examples specific to our domain, along with an issue we discovered.

5.6 Modeling and Analyzing Security

The architectural model documents system-wide design decisions and constrains the detailed design and implementation of system components. Typical design constraints include the input and output associated with a component (e.g., the interface descriptions), data types, connections among components, the mechanisms and protocols by which data and control are exchanged, the use of shared resources, and so forth. The components must, of course, comply with the architectural specifications for the designed properties to hold in the implemented system.

This is the case for architecture design, implementation, and analysis in general. To determine whether security requirements are being met, the model must be annotated to capture security characteristics so that the system can be analyzed appropriately. In our approach, we developed security properties based on security threat scenarios as described in Section 4 and constructed logic rules within Resolute that can be run against the model to verify that these security properties are satisfied by the architecture.

A number of known compromises exploited the connection between the entertainment system and the CAN bus shown in Figure 4. As described in Section 5.2, we can create a trust boundary to mitigate this threat. Assume that the CC components could be compromised and, in particular, that a component is tampering with the speed sensor data.

A tactic to address this problem is to specify authentication/authorization mechanisms in the architecture, to associate group memberships with vehicle components, and then to ensure that data can only be accessed by members of the appropriate group, given appropriate permissions.

Here are the steps necessary to develop and evaluate security characteristics against a model expressed in AADL:

1. Ensure that the necessary architectural components are represented in the model.
2. Express the conditions that must exist to ensure the architectural components meet a predefined security claim (rule).
3. Define the properties that capture the characteristics of the security rule to be verified.
4. Annotate the model problem with the security property (or properties) for checking the security rule.
5. Compose the security rules in the Resolute model checker.
6. Run the Resolute rule checker on the model implementation, and observe the outcome of the rule evaluation.

Next, we walk through our realization of these steps for two scenarios that cover the most important security requirements for the CC infotainment center example: counteracting EoP and establishing groups of trusted components.

5.7 Scenario 1: Architectural Analysis to Counteract EoP Attacks

5.7.1 Architecture of the Original System

In Appendix B, we provide the complete model problem constructed for this work. In this section, we describe a subset of the components we can use as a basis for our scenarios. We choose a subset so that it is small enough to discuss and graphically display in this description. The subset of the vehicle electronics that we present here includes sensors (WheelRotation, BrakePedal), actuators (ThrottleActuator), and control application software (CC and infotainment). The CC system reads vehicle speed from the wheel rotation sensor and compares it to a speed setpoint. Then, the CC system outputs a control variable to the throttle actuator that is proportional to the delta between the desired speed and vehicle speed. The brake pedal system also communicates its status to the CC system. Using the brake pedal causes the CC system to reduce the throttle signal to zero. The infotainment system has multiple modes of operation including one in which it displays various vehicle state variables such as the vehicle speed. The graphical representation of the logical view (e.g., component and connector view) is shown in Figure 12.

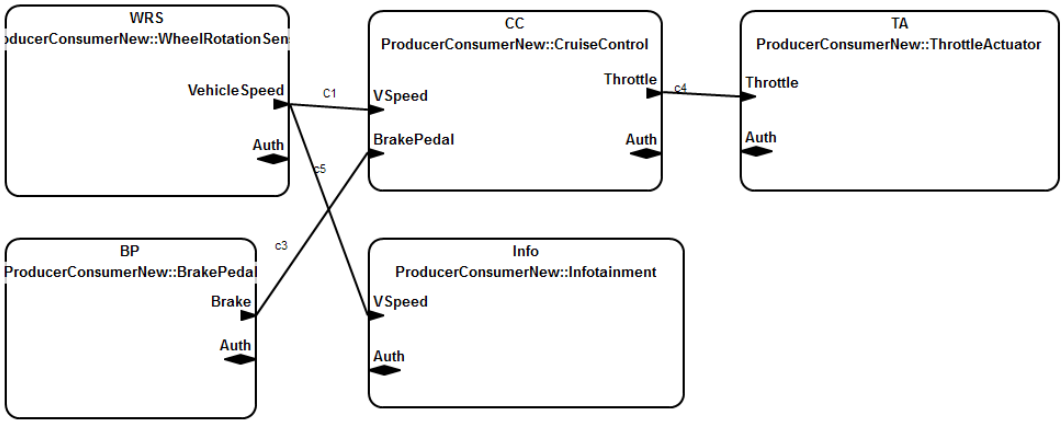

Figure 12: Component and Connector View of a Subset of the Model Problem

5.7.2 Architecture of the System Including Authentication

To provide support for tampering prevention, two techniques are employed. The first is an architectural change that ensures every component is authenticated before it can provide or acquire data in the system. In Section 5.7.3, we discuss the second technique, which employs the concept of data access privilege. The details of how the authentication works are not described in the architecture description. However, the components are annotated with user-defined properties that specify authentication properties. A representation of the authentication server and associated communication connections is shown in Figure 13.

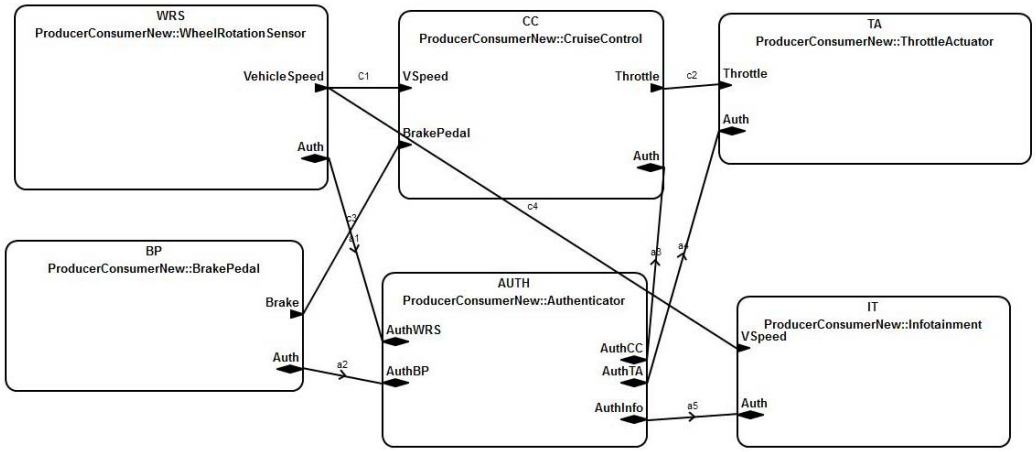

Figure 13: Subset of Vehicle Control System with Authentication Server

Figure 13 shows the logical connections between the authentication server and the components with which it communicates. As part of the architectural description, is it necessary to specify information about the server that can be used for evaluating system security, such as the contents, type, units, associated encryption techniques, and the directionality of the communication. A corresponding physical view of this system would show the hardware communication paths, execution platforms, and the mapping of the software functionality to them. To these components, the architect would specify bus communication protocols, bus bandwidth capacity, the budget of each communication activity, and so forth. Such specification would allow the assessment of security

measures together with the cross-cutting effects of this security, such as performance consequences.

5.7.3 Semantics of Access Privilege

A second technique employed to mitigate EoP is access privilege. Given a piece of data, we might want to constrain how that data can be used and which components can access it. We describe a data access design modeled after the UNIX file protection scheme where files contain access mode designations that specify if the data is read-only, write-only, modifiable (read-write), or executable (a command or series of commands). Each of these modes has an associated group accessibility that indicates which groups can access the data the way the mode allows—that is, which groups can read the data; write it; read and write it; or execute it. Determining, in the architecture, who can access which data is important in providing insight into overall system security. In subsequent sections, we show how this access model can be specified and analyzed in AADL.

5.7.4 Definition of Properties for Access Right Privileges

We need to translate the access model described above into analyzable pieces of information that can be placed into an AADL specification. We created a user-defined AADL property named AccessMode that specifies one of the following enumerations: *r* for read access, *w* for write access, *rw* for read-write access, and *x* for execution permission. Associated with that data is some stipulation about which entities can access that data. For example, a vehicle speed sensor can write to a global data area where multiple control systems in a vehicle can read the vehicle speed. The data could be tagged with an access property of *r* and a list of subsystems that are allowed to read it; for example, the CC system. A malicious application could attempt to write to the vehicle speed location to overwrite the real speed value with a much lower value. That lower value would, in turn, cause the CC component to increase the amount of fuel to the throttle actuator/fuel injectors, resulting in rapid acceleration of the vehicle to reach a high speed. Hence, in this example, we need to specify that only certain applications can write to the speed location and only certain ones can read the speed data. To accomplish that specification, we created another user-defined property named AccessGroup that would contain the list of applications that can access the data.

Finally, we created a user-defined property named AccessProtection that is a list of records containing the AccessMode and the AccessGroup properties.

We formally specify the AccessProtection, AccessMode, and AccessGroup properties in a property set description within our model as follows:

```
property set Security_Trust is

-- properties to support documenting and analyzing security
-- Added property that supports access mode of data

   AccessProtection: list of record (
      AccessMode: enumeration (r, w, rw, x);
      AccessGroup: enumeration (CC, ABS);
   ) applies to (all);

end Security_Trust;
```

The user would then annotate a specific data type with the AccessProtection properties in the following way:

```
data VehicleSpeed
properties
       Security_Trust::AccessProtection => ([
          -- the Cruise Control has R access
          AccessMode => r;
          AccessGroup => CC;
          ],
          -- the ABS has read write access
          [AccessMode => rw;
          AccessGroup => ABS;
          ]);
end VehicleSpeed;
```

In this example, the CC system can read and write VehicleSpeed data, but the anti-lock brake systems (ABS) can only read it.

From the AADL perspective, defining properties in a record is valid and seems to be the best way to express the AccessProtection properties Since Resolute can't parse a record for individual properties, we recast the user-defined properties as AADL enumerations. We believe the general concept would be captured by a single record containing both the AccessMode and AccessGroup properties.

The revised property set becomes

```
   AccessModeNew: enumeration (r, w, rw, x) applies to (all);
   AccessGroupNew: enumeration (cc, abs) applies to (all);
```

And the revised example now becomes

```
data VehicleSpeed
properties
       -- the Cruise Control has Read access
       Security_Trust::AccessModeNew => r;
        Security_Trust::AccessGroupNew => CC;
end VehicleSpeed;
```

Given these properties, we now need to formulate the Resolute rules that will be used to check the architecture to ensure that the data can be accessed properly and only by the allowed components.

5.7.5 Development of Resolute Claims for Access Privilege Compliance

Earlier in this section, we informally described some guidelines about how data can be accessed and by which components. We need to revisit those guidelines in the context of AADL components and their associated semantics so the checks can be explicitly formulated in Resolute. Recall that components interact with other components in AADL through event and data ports. Ports provide the interface specification of connection characteristics (e.g., queued, not queued) and, optionally, the data that is communicated via the port. Consider a simple example where a WheelRotationSensor component with an output data port is connected to the input port of a CC application. The semantics of this configuration is that the WheelRotationSensor writes a VehicleSpeed data value to the output port that is then read via the input port of the CruiseControl component. At this level of specification, the method of data exchange can be ignored. The

CruiseControl application, through its input port, will read the vehicle speed data. The method of the component's data access is specified by the directionality of the port. The semantics of the port specifier in AADL allows for ports to be specified as output, input, or bidirectional. By definition, a component that has an input data port can only read that port. Similarly, a component that has an output port can only write to that port. Components that have bidirectional ports can both read and write to that port. From this observation, we can formulate a rule that will check the AccessModeNew of data attached to a port. Data associated with an input port should have read access, data associated with output ports should have write access, and data associated with bidirectional ports should have read-write access. (We will defer checking and specifying execution privilege until later in this section.)

To check that the architectural description satisfies the design intention, we write claims in Resolute that ensure components have input, output, or bidirectional data ports and that data specified with those ports have the correct AccessModeNew property. Resolute rules are contained in a separate AADL package. In our case, the package name is SecurityCase, and the text for the three claims is shown below.

```
package SecurityCase
    -- Claim to check read privilege on incoming data
    -- making use of the fact that an incoming port with data is a feature of
the component and that
    -- component will be reading that data

SC_ReadPrivilege(self:component) <=
    **  "Read privilege is matched on component " self **

    forall (comp : component) (f : features (comp)). ((f instanceof
data_port) and (direction(f) = "in") and has_type(f))
    =>
-- if (property (type (f) , SecurityProperties::AccessProtectionNew) = "r")
then true else false
    property(type(f), SecurityProperties::AccessProtectionNew) = "r"

    -- Claim to check write privilege on outgoing data
    -- making use of the fact that an out port with data is a feature of the
component and that
    -- component will be writing the data
    SC_WritePrivilege(self:component) <=
    **  "Write privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "out") and has_type(f))
    =>
-- if (property (type (f) , SecurityProperties::AccessProtectionNew) = "r")
then true else false
    property(type(f), SecurityProperties::AccessProtectionNew) = "w"

    -- Claim to check read write privilege on incoming data
    -- making use of the fact that an in-out port with data is a feature of
the component and that
    -- component will be reading and/or writing the data

    SC_ReadWritePrivilege(self:component) <=
```

```
    **  "Write privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "inout"))
    =>
    if ((property (type (f) , Security_Trust::AccessProtectionNew) = "w" )=>
true) or
    ((property (type (f) , Security_Trust::AccessProtectionNew) = "r" ) =>
true) then true else false

end SecurityCase;
```

Claim SC_ReadPrivilege checks that, for any input port with an associated data type, the data has an AccessModeNew property of *r*. Claim SC_WritePrivilege checks that, for any output port with an associated data type, the data has an AccessModeNew property of *w*. In a similar fashion, claim SC_ReadWritePrivilege checks any bidirectional port to ensure that the associated data type has an AccessModeNew property of *rw*.

Accommodating the AccessModeNew that indicates execution is a bit more challenging. The semantics of executing data means that the "data" could comprise a series of instructions (possibly with self-contained data) that are to be executed. In such cases, there is something that interprets the instructions as commands. For example, in the UNIX/LINUX operating system, the shell is a command interpreter. It parses each command keyword, checks if the command matches a name in its command library and, if it does, loads the command image into memory and executes it. At the assembly language level, the interpreter could be a small program that places the command data into memory and then forces the CPU to begin execution at the address where the command data is located (e.g., load the instruction register with the first word of the data and execute it). In either case, whatever mechanism is implemented to force the CPU to fetch/decode/run a command is contained within the system type abstraction. It may or may not be explicitly modeled. A reasonable way to capture this capability is to create a user-defined property named CmdExecution that can be set to true if a command execution capability is intended. A rule would check if the AccessModeNew property of data associated with an input port is *x* and then check the component's CmdExecution property to see if it is *true*. If so, the claim would be proven. The Resolute formulation of the claim to check execution privilege is shown below:

```
       -- Claim to check execution privilege on incoming data
       -- If data being read by component is a command, additionally check
that component has
       -- Command Execution capability, set by CmdExecution Property

       SC_ExecutionPrivilege(self:component) <=
       **  "Execution privilege is matched on component " self **
       forall (comp : component) (f : features (comp)). ((f instanceof
data_port) and (direction(f) = "in"))
       =>
       if ((property (type (f) , Security_Trust::AccessProtectionNew) = "x")
       and
       (property (type (f), Security_Trust::CmdExecution) = true)) then true
else false
```

Having created user-defined properties for EoP attacks and the Resolute rules to check compliance to the rules in the architecture, we can now illustrate the rules' use on a subset of our model problem.

5.7.6 Architecture Components Annotated with Properties

So far, we have identified the attributes of the AADL components that we need to check for, encoded them into a user-defined property set, and formulated the Resolute claims to check all the components in the architecture. The next step is to annotate the components of our model problem with the appropriate properties, specify the claims we want to verify, and run the Resolute model checker.

First, we look at data components and decide which values from the AccessModeNew and AccessGroupNew properties each data component in the system should have. Then, we specify the data with the port classifier for each component in the system. Where appropriate, we specify the CmdExecution property for components that are intended to have command execution capability. Figure 3 on page 8 represents the subset of components used in the system ExampleCC: WheelRotationSensor, ThrottleActuator, and the CruiseControl system. The following AADL text depicts the VehicleSpeed data and the Throttle Position data associated with the appropriate ports. We show the complete model in Appendix C.

```
system BrakePedal
   features
      Brake: in out data port BrakeData;

end BrakePedal;

system ThrottleActuator
   features
      Throttle: in data port ThrottlePosition;
end ThrottleActuator;

system CruiseControl
   features
   VSpeed: in data port VehicleSpeed;
   Throttle: out data port ThrottlePosition;
   Auth: in out data port AuthData;
    BrakePedal: in data port BrakeData;
    properties
     Security_Trust::AccessGroupNew => cc;
     Security_Trust::CmdExecution => true;
end CruiseControl;


system Infotainment
   features
   VSpeed: in data port VehicleSpeed;
   Auth: in out data port;
end Infotainment;


system ExampleCC
end ExampleCC;

system implementation ExampleCC.impl
   subcomponents
```

```
        WRS: system WheelRotationSensor;
        CC: system CruiseControl;
        TA: system ThrottleActuator;
        Info:system Infotainment;
    BP: system BrakePedal;
    connections
        c1: port WRS.VehicleSpeed -> CC.VSpeed;
        --c4: port CC.Throttle -> TA.Throttle;
        c5: port WRS.VehicleSpeed -> Info.VSpeed;
        c3: port BP.Brake -> CC.BrakePedal;
    properties
        Security_Trust::CmdExecution => true;
annex Resolute {**
    prove (SC_ReadPrivilege(this))
    prove (SC_WritePrivilege(this))
    prove (SC_ExecutionPrivilege(this))
**};
```

5.7.7 Running the Resolute Model Checker

Running the Resolute model checker within OSATE[18] requires that the model be instantiated. Resolute walks the instantiated model hierarchy, looking for components specified in its claims and then checks those components according to the logic encoded in the claim.

In this example, we execute the read, write, and execution privilege checks over the entire model. For the read privilege, we want to ensure that data being read by the CC system has the read mode specified and likewise for the write mode. For example, if the command execution check determines that the data is specified as executable, does the CC system have that capability? Figure 14 shows the results of running the checks.

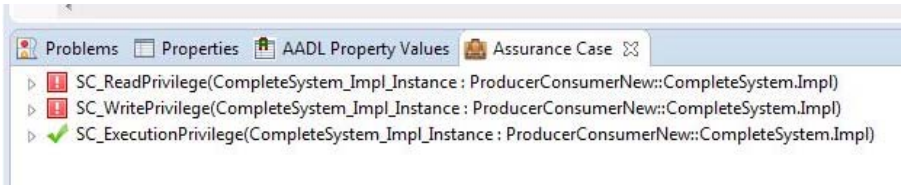

Figure 14: Resolute Results

The read and write privilege checks both failed, but the execution check passed. The AccessModeNew property was set to *w* on the incoming port for VehicleSpeed data and *r* for the ThrottlePosition. As a result, both checks failed. The BrakePedal position data was marked as executable, and the CC system's execution property was set to *true*, so the claim passed. While this example shows the functional viability of the rule checking, it also shows that this reduced scope is not sufficient for adequately specifying and checking the entire system. Brake data could potentially go to many different components, and the access modes for that data must be specified for all those components.

[18] OSATE is an open-source tool platform that provides end users with a complete text editor and a simple analysis tool for AADL and provides developers with full support for the AADL meta-model on an Eclipse platform.

5.8 Scenario 2: Ensuring the Level Among Components

To avoid unnecessary clutter in this description, we use only small subsections of the entire model and show how all the steps apply. When the Resolute engine is run, it traverses the entire model, evaluating the expressed claims on every model component.

5.8.1 Express the Conditions for the Security Rule to Be Developed

Consider the case where one component, the producer, has a logical connection to another component, the consumer. The producer has a port interface specification where data is made available to the input data port of the consumer. A trust boundary exists between the producer and consumer. Two conditions must be satisfied for the model to represent the desired trust:

1. Both the producer and consumer components must belong to the same group.
2. The trust level of the consumer (receiving) component's data must be at the same or greater level of trust as the producer (sending) component's data.

Both of these rules can be written as Resolute claims and checked in the architecture.

5.8.2 Define the Properties in AADL for Trust-Level Assurance

Following the development of Scenario 1, we can capture the notion of group membership and trust level in the form of AADL properties. We must specify the properties in a property set description within our model, as done in the following code.

```
property set Security_Trust is

-- properties to support documenting and analyzing trust boundaries in a
system
-- Trust level {Low, Medium, High} - initial cut at trust levels
-- Constants defined for our TrustLevel Property, with the semantics
-- that the higher the value, the greater the trust level
Low : constant aadlinteger => 0;
Medium : constant aadlinteger => 1;
High : constant aadlinteger => 2;

-- Definition of the TrustLevel property
TrustLevel: aadlinteger applies to (all);

-- Specify the allowable groups in the GroupMembership property
  GroupMembership:  enumeration (StabilityControl,
CruiseControl,DriverDisplays, DriverControls) applies to
        (device, system, process, thread, processor,memory,bus);
end Security_Trust;
```

The Security_Trust property set describes two properties that can be applied to the architecture components. We define one property, TrustLevel, of AADL integer type that can be applied to any AADL component. To capture the notion of levels of trust, we define numeric constants—Low, Medium, and High—that convey degrees of trust. The numeric relationship among the properties will be defined in the Resolute claim discussed later in this section.

The second property we define is GroupMembership, an enumeration type consisting of components in our model problem, specifically StabilityControl, CruiseControl, DriverDisplays, and DriverControls.

5.8.3 Developing and Annotating the Model Problem for Trust-Level Compliance

The textual model below shows how we applied these properties to our model problem.

```
data WheelRotationSpeed
     properties
       Security_Trust::TrustLevel => Security_Trust::med ;
end WheelRotationSpeed;

data WheelRotationSpeed2
     properties
       Security_Trust::TrustLevel => Security_Trust::low ;
end WheelRotationSpeed2;

device WheelRotationSensor
  features
    output: out data port WheelRotationSpeed
  properties
    Security_Trust::TrustLevel => Security_Trust::low;
    Security_Trust::GroupMembership => CruiseControl;
end WheelRotationSensor;

system CruiseControl
  features
    input: in data port WheelRotationSpeed;
   properties
    Security_Trust::TrustLevel => Security_Trust::low;
    Security_Trust::GroupMembership => CruiseControl;
end CruiseControl;
```

This model contains two data components, WheelRotationSpeed and WheelRotationSpeed2. The intention is to represent two data types with the only difference being the values of TrustLevel, which will become significant shortly.

Two component types are defined as well: the WheelRotationSensor device type and the CruiseControl system type. WheelRotationSensor contains an output port with the associated data type WheelRotationSpeed, and the WheelRotationSensor device has TrustLevel and GroupMembership properties associated with it. The CruiseControl system type has an input data port containing WheelRotationSpeed data. In the composition of the model, the intention will be to connect the output data port of the WheelRotationSensor to the input data port of the CruiseControl system. Both the WheelRotationSensor device type and the CruiseControl system type contain the appropriate GroupMembership property.

5.8.4 Compose the Resolute Rules to Verify the Trust Level

To check that the architectural description satisfies the design intention, we write claims in Resolute that will ensure components connected together with the specified trust levels meet the rules

described above. Resolute rules are contained in a separate AADL package called SecurityCase. The text of this package is shown below.

```
package SecurityCase
-- Contains the Resolute Claims for Security Trust model
public
annex Resolute {**

--
-- Rule SC1 checks that the data component on sending side of a logical
connection has a trustlevel equal to or greater than that of the data
component of the receiving component.
--
    SC1(self:component) <=
    **  "Confidentality security level is met on " self **
    forall (c1 : component) (conn : connection) (c2 : component) . (connected
(c1, conn, c2)) and (has_property (c1, Security_Trust::TrustLevel)) and
(has_property (c2, Security_Trust::TrustLevel))
    =>
    property (c1, Security_Trust::TrustLevel) <= property (c2,
Security_Trust::TrustLevel)

-- Rule SC1a checks if the components on either side of the logical
connection have the same Group Membership
    SC1a(self:component) <=
    **  "Group membership criteria is met " self **
    forall (c1 : component) (conn : connection) (c2 : component) . (connected
(c1, conn, c2)) and (has_property (c1, Security_Trust::GroupMembership)) and
(has_property (c2, Security_Trust::GroupMembership))
    =>
    property (c1, Security_Trust::GroupMembership) = property (c2,
Security_Trust::GroupMembership)

-- Rule SC1s is the logical and of security rules SC1 and SC1a e.g. that
each component belongs to the same group and the data confidenatlity is
maintained between the components.
    SC1c(self:component) <=
    **  "Composite Check_ " self **
    SC1(self) and SC1a(self)
**};
end SecurityCase;
```

We use the claim composition capability of Resolute and write two rules to check each sub-condition (SC1 and SC1a) and then combine the rules into one larger rule (SC1c). Rule SC1 verifies that two connected components must be in the same group. Rule SC1a verifies that the trust level of the data associated with the output data port of one component is equal to or less than the input data port of the other data component. For example, the data trust level of the writing component must be less than or equal to the trust level of the reading component's data. Rule SC1c checks that both rules are true for a valid trust-level specification.

5.8.5 Compose an Implementation of the Model and Run Resolute

In the preceding sections, we annotated the architectural components in our model with the TrustLevel properties so the TrustLevel rule SC1c could be evaluated. To provide the context for rule evaluation in AADL, we must first compose our components into the logical structure of our model and then run the Resolute rule engine over that model. Below is the AADL model that contains the specification of the WheelRotationSensor device communicating the WheelRotationSpeed to the CruiseControl system (the software component that reads the wheel rotation speed and compares the speed to some predetermined speed setpoint).

```
system CompleteSystem
end CompleteSystem;

system implementation CompleteSystem.Impl
  subcomponents
    WheelSpeed: device WheelRotationSensor;
    CruiseControlApp: system CruiseControl;
  connections
    c1: port WheelRotationSensor.WheelRotationSpeed ->
CruiseControlApp.CruiseControl;
annex Resolute {**
   prove (SC1c(this)) -- Prove the complete TrustLevel Rule
   prove (SC1(this))  -- For illustrative purposes, prove each subrule
   prove (SC1a(this))
**};
end CompleteSystem.Impl;
```

Figure 15 shows what our model looks like graphically.

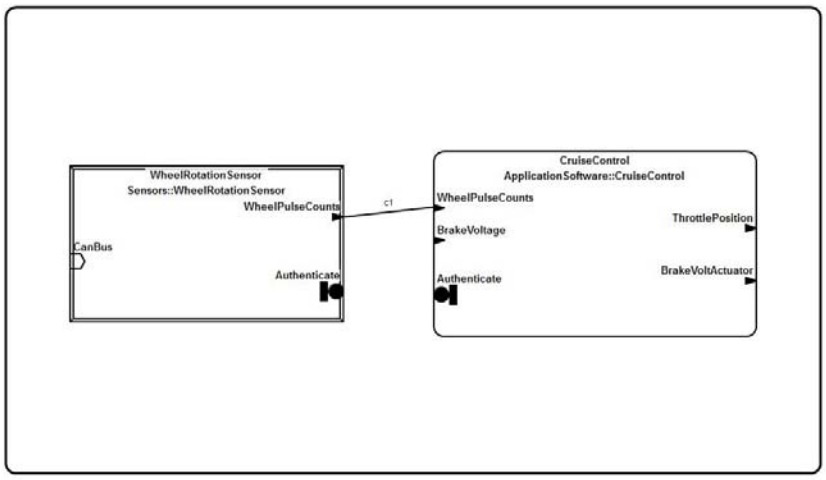

Figure 15: AADL Model Showing WheelRotationSensor Connected to CruiseControl

The subcomponents sub-clause above contains the components to be contained in the implementation of our model. The connections sub-clause specifies the connections between each of the specified subcomponents. The annex Resolute {** **} sub-clause contains the names of the Resolute rules to be checked against the implementation; for example, CompleteSystem.impl together

with the keyword *prove*. The keyword *this* sets the context for the evaluation, specifically the implementation that contains the Resolute annex, CompleteSystem. In this example, the rules to check are SC1, SC1a, and SC1c, contained in the SecurityCase package.

As mentioned earlier, rule SC1c is the conjunction of rules SC and SC1a. In the current version of OSATE (2.0.9), running the check on all three rules produces the output shown in Figure 16.

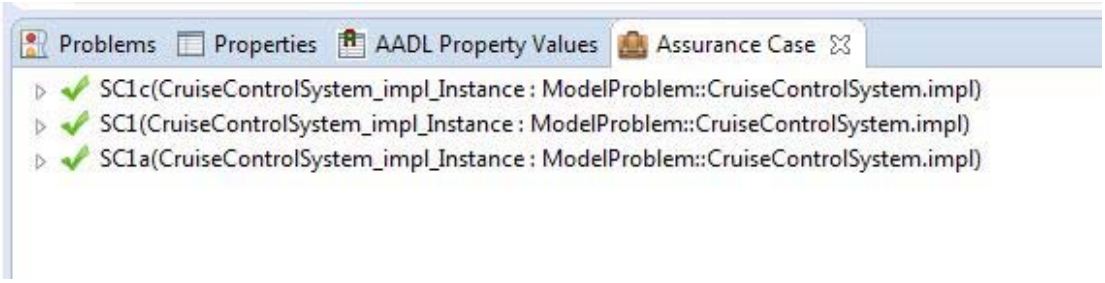

Figure 16. Resolute Rule Check Results Showing TrustLevel Rule SC1c and Its Two Component Sub-Rules, SC1 and SC1a

Since SC1 and SC1a both pass, SC1c should also pass. We can now change the GroupMembership of WheelRotationSensor to ABS, which will cause the rule check to fail, as shown in Figure 17.

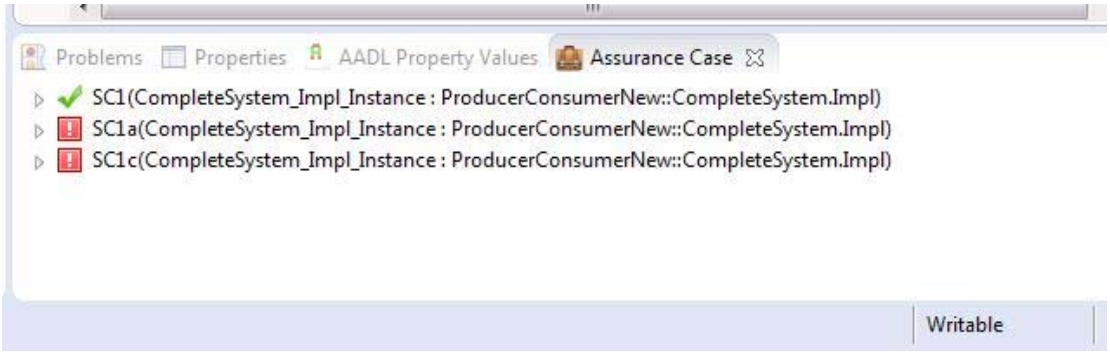

Figure 17: Resolute Rule Check Results Showing the Failing of the TrustLevel Rule SC1c and SC1a and the Passing of SC1

The passing of the rules indicates that the architectural specification meets the conditions to ensure the TrustLevel rule between the two components. This means that the implementation of the actual software should contain TrustLevel and GroupMembership parameters with the correct values assigned to each.

Now it is straightforward to extend this small example to a chain of components that represent the entire logical data flow; for example, WheelRotationSensor – CruiseControl – ThrottleActuator and assigning GroupMembership and TrustLevel to the CruiseControl output data and the ThrottleActuator input data. Applying the Resolute rule checks would evaluate each interface connection along the chain and report the results accordingly.

5.9 Summary

As our example shows, the use of user-defined properties can effectively express key security characteristics of a system. Annotating a model component represents a constraint on downstream development activities. We specify the characteristics that implementations must contain so that the security approach can be verified over the entire architecture. The properties have values that can be checked across the architecture to ensure that a security concern can be validated. This check implies that the implementation must properly implement these properties.

For our example, Resolute provided an effective solution to the analysis capability. The issue of not being able to understand properties in a record structure is a limitation that affected the usefulness of the approach, but we did find an acceptable workaround.

Given that the security characteristics can be captured and analyzed allows us to explore the effects of the interaction with other quality attributes across the entire system well before implementation and integration.

6 Conclusions

In this report, we have shown how selected security properties can be modeled architecturally using AADL. We exemplified this process in a cyber-physical example—that of automotive electronics—because it is a timely and relevant example with enormous real-world consequences. Future work is needed to show how additional threat categories can be addressed using AADL.

As we have learned, doing this architectural security modeling has several concrete benefits. First, it guides an architect to think in terms of system-wide security properties and how they can be established, expressed, and maintained. This activity actually changes the process of design, which needs to begin with threat modeling—in our case, using the STRIDE framework. The designer uses the threats identified in the threat model to guide the architect in establishing design strategies to mitigate the risks imposed by the threats. This mitigation may involve the creation of new components (such as an authentication service), new connections, and new properties. By modeling the architecture in this way, we can ensure that any decisions made are consistent with the system-wide strategies that have been designed.

Second, this approach provides a framework for checking that such properties are maintained during system maintenance and evolution. This checking, of course, assumes that the architectural model and the implementation are maintained in lock-step. That is, when code changes are made, any architectural implications of those changes are reflected in an updated AADL model, and if the model changes, we can be immediately alerted to any modifications that must be made in the implementation.

However, this approach is not a panacea. Next we discuss some of the limitations associated with architectural modeling for security.

6.1 Limitations

Scenarios 1 and 2 in Section 5 used security policies, as specified in the AADL model, to guard against the identified security threats. AADL modeling provides an effective way to document and verify security policies across a system or indeed across multiple systems. But because the AADL model does not *validate* that those policies provide sufficient security, we need to independently determine whether the enforcement of those polices sufficiently secures the system, given its planned usage. The problem here is not with AADL, or with any such modeling approach, but rather with the limitations of modeling the adversarial aspects of security. Attackers often use the preliminary phases of a compromise to build their own model of a site's security architecture. Attackers look for ways to invalidate the very assumptions underlying the security architecture. For example, security attacks such as an application buffer overflow or successful phishing of a user can allow an adversary to obtain network privileges without authentication.

Furthermore, the verification offered by tools such as AADL (and its associated tools, such as Resolute) does not show that the specifications provide the *desired* level of security assurance. For example, an architect could claim that a component sufficiently verifies input data to eliminate the risk of a SQL injection. However, such a claim is difficult to validate for SQL injections, and alternative mitigations might in fact have higher levels of assurance.

We can improve security (and reliability) by tolerating an event at runtime or by employing design strategies that reduce or eliminate the possibility of such an event. The AADL appendix on fault modeling enables an architect to specify how a system detects, reports, and responds to runtime errors [Delange 2014]. Fault tolerance for hardware failures can draw on well-understood mitigations such as the use of redundant servers. We have much less experience with tolerating software defects. We may be able to detect a software error, but recovery strategies might depend on distinguishing an error caused by normal but unanticipated conditions from one where the failure was orchestrated by an adversary. Currently improved security and software reliability depend much more on fault prevention than fault tolerance.

Appendix A: Threat Modeling Using the Elevation of Privilege Game

Elevation of Privilege (EoP) is a card game designed to draw people who are not security practitioners into the craft of threat modeling [Shostack 2012]. Because it does not require detailed design, we found it useful to generate a set of potential threats faced by our model system. This appendix provides an overview of the game, describes our experience using it, and provides some observations about the process.

About the Game

We provide a brief summary of the game here; Shostack describes it in more detail [Shostack 2012]. The game consists of 74 playing cards, plus an additional 2 instruction cards, 6 reference cards, a flowchart card, and an "about" card. The playing cards are in six suits, each representing one of the concepts in the STRIDE model: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Each card suggests a threat that might affect the system being modeled. Suits are arranged in approximate order from least to most impactful threats, with the ace cards representing new and novel threats invented by the players. The only prerequisites for playing the game are (1) a diagram of the system to be modeled, (2) a way to track the problems identified (an easel pad and markers worked well for us), and (3) a deck of EoP cards.

Playing the Game

We were interested in identifying threats relevant to the automotive system that forms the guiding example of this report. In the context of the game, we used the diagram shown in Figure 18 on page 50 to guide the discussion. Note that while we wanted to reflect a sufficiently realistic system, we were not attempting to describe any specific existing design. Nor do we claim any expertise in automotive system engineering. Thus, the diagram is intended to be a representative design rather than a specific real-world one. See Miller and Valasek's work for more detailed system descriptions [Miller 2013, 2014, 2015].

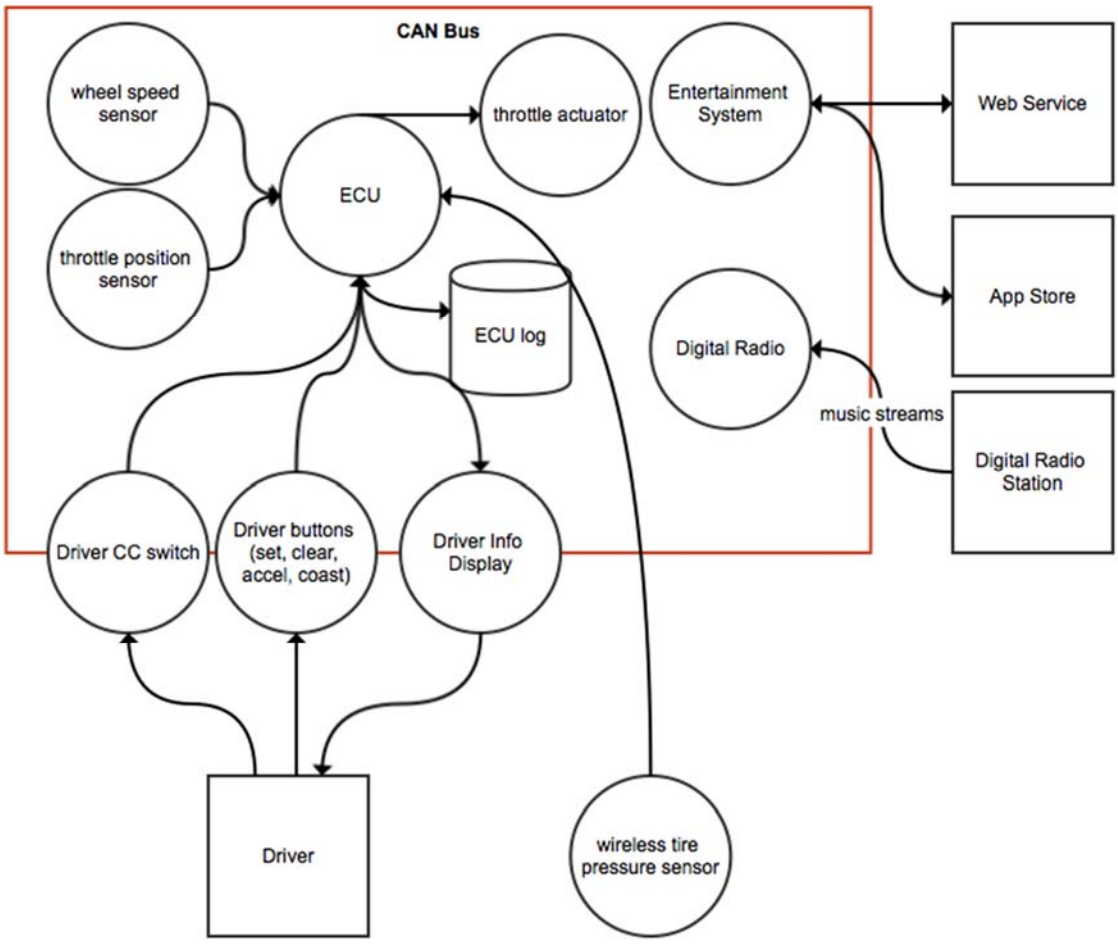

Figure 18: Model Automotive Cyber-Physical System

We played the game twice, with two distinct groups of five players:

1. The first game was played by vulnerability analysts experienced in discovering, identifying, and analyzing software vulnerabilities.
2. The second game was played by security and software engineering researchers with a broader scope of experience than the first-game players.

The diagram evolved somewhat during the game play, as it became evident that our initial rough sketch lacked salient details. That evolution in itself, was a useful result of playing the game: We refined our model of the system based on our reasoning about its security properties.

Tables 1 and 2 show the cards played, along with a summary of the threat associated with each.

Table 5: Cards Played in First Game (by Vulnerability Analysts)

Card	Threat
3T	Entertainment system fails to validate SSL certificates.
6T	Attacker can overwrite entertainment system code.
8T	Tire pressure wireless link has no integrity checks so packets can be injected onto CAN bus.
JT	Given compromised entertainment system, an attacker can send throttle control messages to throttle actuator because no ACLs exist on CAN bus.
QR	Tire pressure sensors can be updated with no logs.
KR	The CAN bus has no logging of data sent over it.
5I	User can listen to a subscription-based radio station without paying for it because the satellite radio uses proprietary crypto.
7R	Attacker can saturate the electronic control unit (ECU) log ring buffer and make it wrap.
8R	Attacker can erase the ECU logs.
JI	ECU firmware has embedded symmetric key to decide upgrade validity (attacker gets key).
KI	Attacker can read data on CAN bus because there is no crypto.
3I	Attacker can capture security-relevant error messages via CAN bus sniffing (e.g., OBD2 port).
JD	Attacker can effectively disable logging on ECU by flooding ECU with bogus messages.
2D	If you drain the battery, the key fob won't open the door (no mechanical key or externally accessible lock).
5D	Attacker floods the driver information system with messages to prevent it from updating the speed indicator.
AD	Attacker can cause arbitrary information to be displayed (rather than what should be).
7D	Attacker physically smashes the driver information system.

Table 6: Cards Played in Second Game (by Security and Software Researchers)

Card	Threat
3T	Attacker can leverage unchecked certificate against entertainment system and send arbitrary messages on the CAN bus.
4T	Entertainment console makes decisions about what it communicates to the driver console and how it checks error-handling messages (valid source?).
E9	Driver information display doesn't know whether the ECU checked the validity of wheel speed.
KT	Attacker can load code to ECU via OBD2 port, exploiting other services.
QS	Physical access to car lets you alter the account information with remote vehicle assistance service provider.
KS	Stored default admin password on entertainment system does not force change on first use.
AS	ECU manufacturer embeds persistent code in supply chain.
1S	Attacker can spoof crash-alert messages by exploiting a vulnerable component on the CAN bus.
JS	XSS on entertainment system could steal login credentials and vehicle identification number (VIN) (VIN used as authenticator?) from OnStar device.
8S	Attacker breaks into dealership, steals OnStar credentials, and shuts down many cars.
3S	Attacker does a remote start by spoofing or breaking by brute force the keyless entry codes.
5S	During Bluetooth-pairing process, an attacker could spoof the device being paired to the radio.
5R	Attacker can modify log messages in transit on the CAN bus because of weak integrity controls.
KR	Most components have no logs.
9R	Entertainment system can falsify log entries as if they come from the radio.
3R	ECU logs security relevant information that can be copied out via OBD2 port (unauthenticated).

Discussion of Game Mechanics

We can confirm that five players were reasonable for the game. Had the group been much larger, it seems unlikely that the game play would have made it around the table enough times to get input from all the participants. We played for about 90 minutes in the first game and 2 hours in the second. In each case, that was enough time to get through four tricks (one trick = one time around the table), plus a bit of meta-discussion about the game itself.

We attribute the quicker play of the first game to these player qualities: (1) they were better acquainted with the system being modeled and (2) they were experienced vulnerability analysts who were already accustomed to thinking about systems from an attacker's perspective and had prior exposure to threat modeling techniques. However, given the choice, two hours is a better timespan, especially if players need an overview of the system and the game.

As for the game play, we observed that the scoring incentivizes the playing of high cards early in the game, which naturally imposes a greedy knapsack heuristic [Dantzig 1957] for threat prioritization. That was a nice benefit.

Playing an ace can slow the game down due to debate over whether the vulnerability identified is really unique compared to all the other cards in the suit. Shostack notes this as well:

> *"This discussion of equivalency between threats distracts somewhat from game play, but offers an opportunity for a deeper discussion of the suits and threat equivalency"* [Shostack 2012].

We found the discussion to be useful though and couldn't find much fault in the game for the interpretation of aces. In the future, if a quicker pace of play is needed, we might consider using the aces as a wildcard for the suit topic instead. However, that would of course alter the effectiveness of the scoring method recommended in the game.

There was an interesting dynamic among playing the game for its own sake, identifying vulnerabilities, and understanding the implications of how the game worked. For example, from the perspective of identifying vulnerabilities, it seemed odd to play a card without identifying a threat and still be eligible to win a point for the trick. The instruction card says "Each round is won by the highest card played in the suit that was led, unless an Elevation of Privilege (EoP) card is played." In the ensuing discussion, we concluded that a useful variation could be to require the player to identify a threat to be eligible to win the trick. (i.e., the highest card played with an associated threat identified is the winner of the trick).

Only after playing the game did we notice the following text in Shostack's paper:

> *"The written rule says to only count the highest card which was actually connected to the system being developed, but in practice this is sometimes discarded to give a deeper involvement to beginners"* [Shostack 2012].

This text supports our conclusion that experienced analysts may want to play according to the stricter scoring rules.

Some cards were difficult to relate to the automotive system we were modeling and had a distinct "traditional computing flavor." Potential future work could include adapting the deck to threats more specifically focused on IoT devices or CPSs.

During the game, we found it easy to divert into discussions of the game strategy, tactics, and aspects of the game's scoring rules. However, it is important to remember that the purpose of playing the game is not actually the playing of the game—rather it's the structured brainstorming of threats. To that end, these variant rules seem worthwhile, although we did not try them this time around:

- Double the number of points, and give one point for threats on other people's cards.
- Other players may "riff" on the threat, and if they do, they get one point per additional threat.

We had to remind players not to get bogged down in much discussion of "Does the system behave exactly this way?" and instead to simply note the threat and move on. If we were actually responsible for building the system, we would take the list of identified threats and refine it further by matching it up against the actual system to identify mitigating factors, validity of the threat, and so forth.

We also had difficulty because the scope of the system we were modeling wasn't entirely clear. For example, should we have allowed the radio playing unauthorized music to score? We opted to be inclusive, since the threats can always be dropped off the list in a later refinement. Similar to brainstorming, it seemed like the game time was better spent generating ideas than filtering them too harshly.

Our scenario seemed to provide an analog to threat modeling a system in the early stages of development—say, at a point when the system itself is underspecified and ambiguity in its components or behavior is expected. Having an actual expert on the system in the room could alleviate some of the debate on whether a threat is realistic. It might also help to implement the "Are you willing to create a ticket to fix this?" rule. In our case, we weren't responsible for the development of the system and didn't have any resident experts on it. While those caveats may be typical for vulnerability analysts evaluating a third-party system, it is likely a problem for teams attempting to threat model systems they are directly responsible for.

Discussion

While some of the identified threats were based on previously reported vulnerabilities, others would later emerge as actual vulnerabilities in real vehicles. For example, the 8T play in Table 5 on page 51 is equivalent to what Roufa and colleagues describe [Roufa 2010], while Mahaffey describes a remote attack against a Tesla Model S similar to (although not exactly the same as) the Jack of Tampering play also in game 1 [Mahaffey 2015]. There is also overlap between the threats listed in the games and vulnerabilities described by Miller and Valasek [Miller 2015], Checkoway and colleagues [Checkoway 2011], and Koscher and colleagues [Koscher 2010].

When reviewing the summary of the game results, a colleague who had not participated in the game sessions noted that privacy-related attacks didn't appear in either game; for example, the ability of an attacker to remotely enumerate a vehicle's Global Positioning System (GPS) location as Miller and Valasek describe [Miller 2015]. It is unclear whether the absence of that type of attack is an inherent limitation of the players' mindset or of the game and the card prompts. We suspect the issue is with the former, since "Information Disclosure" is one of the suits in the deck.

Process-wise, recording results is critical; someone must be designated as a scribe for the group. He or she can play too but needs to capture the threats identified by the players.

Finally, we recognized that for the effort to be successful, the group must be interested in identifying threats. This game will not work nearly as well if some players have a vested interest in not finding problems. For example, in the case of an adversarial acquisition process in which developers are motivated to have the system accepted by the acquirer, they would have little incentive to identify new vulnerabilities representing additional work that must be completed prior to acceptance.

Conclusion

In summary, we focused our game play on a "connected car" scenario inspired by the Toyota unintended acceleration problem and how it might become a remotely exploitable vulnerability. However, the threats identified were much broader than that. We found the Elevation of Privilege game to be quite useful and plan to keep it in our toolbox for vulnerability analysis.

Appendix B: AADL and STRIDE

The property set below was developed to support the STRIDE model.

```
property set Security_Trust is

-- properties to support documenting and analyzing trust boundaries in a
system
-- Trust level {Low, Medium, High} - initial cut at trust levels
-- Constants defined for our TrustLevel Property, with the semantics
-- that the higher the value, the greater the trust level
Low : constant aadlinteger => 0;
Medium : constant aadlinteger => 1;
High : constant aadlinteger => 2;

-- Definition of the TrustLevel property
TrustLevel: aadlinteger applies to (all);

-- Specify the allowable groups in the GroupMembership property
  GroupMembership:  enumeration (ABS, CruiseControl, DriverDisplays,
DriverControls) applies to
        (device, system, process, thread, processor,memory,bus, data);
-- Group matches to Category in Bell LaPadula confidentiality plug-in


-- Properties that supports access mode of data
   AccessProtection: list of record (
      AccessMode: enumeration (r, w, rw, x);
      AccessGroup: enumeration (CC, ABS);
   ) applies to (all);

---- Since resolute cannot process record properties, I made separate
properties.
-- The down side of this approach is that an object can only have one access
group and access protection
-- Doing these properties only illustrates the concept

   AccessProtectionNew: enumeration (r, w, rw, x) applies to (all);
   AccessGroupNew: enumeration (cc, computer, abs) applies to (all);

 -- Property to support Command execution of a component
  CmdExecution: aadlboolean applies to (all);

-- Data encryption properties - a list of applicable types (these are just
examples, more can be added later)
-- To support Tampering Attack Type

 DataEncryptionHashing: enumeration (MD5, SHA ) applies to (data);
 DataEncryptionPrivateKey: enumeration ( DataEncryptionStandard,
AdvancedEncryptionStandard, InternationalDataEncryptionAlgorithm) applies to
(data);
```

```
 --DataEncryptionPrivateKey: enumeration (DES, AES, IDEA) applies to (data);
 DataEncryptionPublicKey: enumeration (RSA, DiffieHellman) applies to
(data);

-- Timing properties specific to Denial of Service Attacks
-- Interval allowed between incoming messages....if interval exceeded then
trigger DOS alert
    PageRequestTime: Time_Range
       applies to (thread);
-- Maximum amount of time for a page to be serviced
     PageServiceDeadline: Time
       applies to (thread, device, subprogram, subprogram access, event
port, event data port);

    PageExecution_Time: Time_Range
       applies to (thread, device, subprogram, event port, event data port);

end Security_Trust;
```

Appendix C: AADL for Scenarios

The following is a listing of Resolute claims that check for architectural support for trust levels and EoP attacks.

```
package SecurityCase
-- Contains the Resolute Claims for Security Trust model
public
annex Resolute {**

   --SecurityClaim function (SCF1):
--Given a component (e.g., System,Device,Thread) has a property Security
tuple specified (TrustLevel, GroupMembership)
--AND  given an in or out port associated with the component has specified
data classifier,
--Then the Security property tuple of the data classifier must equal that of
the associated component classifier.
--e.g., System (TrustLevel)== port data (TrustLevel) AND
System(GroupMembership) == Port Data (GroupMembership)  then security claim
is valid.
--
-- Algorithm for checking that IF a component (System,Device,Thread) has a
property Security tuple
-- specified (TrustLevel, GroupMembership) and if an in or out port
(associated with the component)
-- has specified data classifier, then the
-- Security property tuple of the data classifier must equal that of the
associated component classifier.
-- e.g., system (TrustLevel)== port data (TrustLevel) AND
system(GroupMembership) == port data (GroupMembership)
   SC1(self:component) <=
   **  "Confidentality security level is met on " self **
   forall (c1 : component) (conn : connection) (c2 : component) . (connected
(c1, conn, c2)) and (has_property (c1, Security_Trust::TrustLevel)) and
(has_property (c2, Security_Trust::TrustLevel))
   =>
   property (c1, Security_Trust::TrustLevel) <= property (c2,
Security_Trust::TrustLevel)

-- Claim to check if the components have proper Group Membership
   SC1a(self:component) <=
   **  "Group membership criteria is met " self **
   forall (c1 : component) (conn : connection) (c2 : component) . (connected
(c1, conn, c2)) and (has_property (c1, Security_Trust::GroupMembership)) and
(has_property (c2, Security_Trust::GroupMembership))
   =>
   property (c1, Security_Trust::GroupMembership) = property (c2,
Security_Trust::GroupMembership)

-- Composit security level and group membership criteria is met
```

```
    SC1c(self:component) <=
    **  "Composite Check_ " self **
    SC1(self) and SC1a(self)


-- Algorithm for checking that IF a component (System,Device,Thread) has a
property Security tuple
-- specified (TrustLevel, GroupMembership) and if an in or out port
(associated with the component)
-- has specified data classifier, then the
-- Security property tuple of the data classifier must equal that of the
associated component classifier.
-- e.g., system (TrustLevel)== port data (TrustLevel) AND
system(GroupMembership) == port data (GroupMembership)
    SC2(self:component) <=
    **  "Confidentality security level is met on " self **
    forall (comp : component) (f : features (comp)). (f instanceof data_port)
    =>
    property (type (f), Security_Trust::TrustLevel) = property (comp,
Security_Trust::TrustLevel)


    -- Rule to check if
    SC3(self:component) <=
    **  "Confidentality security level is met on " self **
    forall (c1 : component) (conn : connection) (c2 : component) . (connected
(c1, conn, c2)) and (has_property (c1, Security_Trust::TrustLevel)) and
(has_property (c2, Security_Trust::TrustLevel))
    =>
    property (c1, Security_Trust::TrustLevel) <= property (c2,
Security_Trust::TrustLevel)

    -- Claim to check read privilege on incoming data
    -- making use of the fact that an incoming port with data is a feature of
the component and that
    -- component will be reading that data

    SC_ReadPrivilege(self:component) <=
    **  "Read privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "in"))
    =>
    if (property (type (f) , Security_Trust::AccessProtectionNew) = "r") then
true else false

    -- Claim to check write privilege on outgoing data
    -- making use of the fact that an out port with data is a feature of the
component and that
    -- component will be writing the data

    SC_WritePrivilege(self:component) <=
    **  "Write privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "out"))
```

```
    =>
    if (property (type (f) , Security_Trust::AccessProtectionNew) = "w") then
true else false

    -- Claim to check read write privilege on incoming data
    -- making use of the fact that an in-out port with data is a feature of
the component and that
    -- component will be reading and/or writing the data

    SC_ReadWritePrivilege(self:component) <=
    **  "Write privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "inout"))
    =>
    if ((property (type (f) , Security_Trust::AccessProtectionNew) = "w")=>
true) or
    ((property (type (f) , Security_Trust::AccessProtectionNew) = "r") =>
true) then true else false

    -- Claim to check execution privilege on incoming data
    -- If data being read by component is a command, additionally check that
componet has
    -- Command Execution capability, set by CmdExecution Property

    SC_ExecutionPrivilege(self:component) <=
    **  "Execution privilege is matched on component " self **
    forall (comp : component) (f : features (comp)). ( (f instanceof
data_port) and (direction(f) = "in"))
    =>
    if ((property (type (f) , Security_Trust::AccessProtectionNew) = "x")
    and
    (property (type (f) , Security_Trust::CmdExecution) = true)) then true
else false
--
**};
end SecurityCase;
```

Bibliography

URLs are valid as of the publication date of this document.

[Almorsy 2013]
Almorsy, M.; Grundy, J.; & Ibrahim, A. S. "Automated Software Architecture Security Risk Analysis Using Formalized Signatures," 662-671. *Proceedings of ICSE 2013*. San Francisco, CA, May 18-26, 2013. IEEE, 2013. http://www.ict.swin.edu.au/personal/jgrundy/papers/icse2013.pdf.

[Asnar 2011]
Asnar, Y.; Paja, E.; & Mylopoulos, J. "Modeling Design Patterns with Description Logics: A Case Study," 169-183. Lecture Notes in Computer Science 6741, 2011.

[AVSI 2015]
Aerospace Vehicle Systems Institute (AVSI). *SAVI – The System Architecture Virtual Integration Program*. http://savi.avsi.aero/ (2015).

[Backes 2015]
Backes, John; Cofer, Darren; Miller, Steven; & Whalen, Michael W. *Requirements Analysis of a Quad-Redundant Flight Control System*. http://arxiv.org/pdf/1502.03343.pdf (2015).

[Bell 1974]
Bell, D. E. & La Padula, L. J. *Secure Computer Systems: Vol. I—Mathematical Foundations, Vol. II—a Mathematical Model, Vol. III—a Refinement of the Mathematical Model* (Technical Report MTR-2547). Mitre Corporation, March–April 1974.
http://www.dtic.mil/dtic/tr/fulltext/u2/780528.pdf.

[Calloni 2011]
Calloni, B.; Camapra, D.; & Mansourov, N. *Volume 2 - White Box Definitions of Software Fault Patterns* (AFRL-RY-WP-TR-2012-0111). Lockheed Martin, December 2011.

[Checkoway 2011]
Checkoway, Stephen, et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces." USENIX Security Symposium. 2011.
http://static.usenix.org/events/sec11/tech/full_papers/Checkoway.pdf.

[Clements 2010]
Clements, Paul et al. *Documenting Software Architectures: Views and Beyond*, 2nd edition. Addison-Wesley, 2010.

[Dantzig 1957]
Dantzig, George B. "Discrete-variable extremum problems." *Operations Research 5*, 2 (1957): 266-288.

[Delange 2014]
Delange, Julien; Feiler, Peter; Gluch, David P.; & Hudak, John. *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment* (CMU/SEI-2014-TR-020). Software Engineering Institute, Carnegie Mellon University, October 2014.
http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=311884.

[Delange 2013]
Delange, Julien. *Simple Version of the ARP4761/AIR6110 Example.*
https://wiki.sei.cmu.edu/aadl/index.php/Simple_version_of_the_ARP4761/AIR6110_example (July 23, 2013).

[FDA 2010]
U.S. Food and Drug Administration. *Guidance for Industry and FDA Staff Total Product Life Cycle: Infusion Pump Premarket Notification [510(k)] Submissions (Draft Guidance).*
http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/UCM209337.pdf (April 23, 2010).

[Feiler 2012]
Feiler, Peter H. & Gluch, David P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[Feiler 2009]
Feiler, Peter; Hansson, Jörgen; de Niz, Dionisio; & Wrage, Lutz. *System Architecture Virtual Integration: An Industrial Case Study* (CMU/SEI-2009-TR-017). Software Engineering Institute, Carnegie Mellon University, 2009.
http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9145.

[Gacek 2014]
Gacek, Andrew; Backes, John; Cofer, Darren; Slind, Konrad; & Whalen, Mike. "Resolute: An Assurance Case Language for Architecture Models," 19-28. *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. Portland, OR, October 2014. ACM, 2014.

[HBR 2014]
Harvard Business Review (HBR). *Internet of Things: Science Fiction or Business Fact?*
https://hbr.org/resources/pdfs/comm/verizon/18980_HBR_Verizon_IoT_Nov_14.pdf (2014).

[Hernan 2006]
Hernan, Shawn; Lambert, Scott; Ostwald, Tomasz; & Shostack, Adam. "Uncover Security Design Flaws Using the STRIDE Approach." *MSDN Magazine*, November 2006.

[Hoglund 2004]
Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code.* Addison-Wesley, 2004. http://www.informit.com/store/exploiting-software-how-to-break-code-9780201786958.

[Howard 2006]
Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Microsoft Press, 2006.

[Howard 2003a]
Howard, Michael, "Fending Off Future Attacks by Reducing Attack Surface." https://msdn.microsoft.com/en-us/library/ms972812.aspx.

[Howard 2003b]
Howard, M.; Pincus, J.; & Wing, J. *Measuring Relative Attack Surfaces*. http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf, 2003.

[ITU 2015]
ITU. "Internet of Things Global Standards Initiative" (2015). http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx.

[Jacobson 1999]
Jacobson, Ivar; Booch, Grady; & Rumbaugh, James. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., 1999 (ISBN: 0-201-57169-2).

[Kelley 2013]
Kelley, Michael B. "The Stuxnet Attack on Iran's Nuclear Plant Was 'Far More Dangerous' Than Previously Thought." http://www.businessinsider.com/stuxnet-was-far-more-dangerous-than-previous-thought-2013-11 (November 20, 2013).

[Kelly 2004]
Kelly, Tim & Weaver, Rob. "The Goal Structuring Notation: A Safety Argument Notation." *Proceedings of International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC 2004)*. Edinburgh, Scotland, May 2004. IEEE Computer Society, 2004.

[Kelly 1998]
Kelly, Tim P. *Arguing Safety*. PhD diss., University of York, 1998.

[Koscher 2010]
Koscher, Karl, et al. "Experimental security analysis of a modern automobile." *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010.

[Kruchten 1995]
Kruchten, Philippe. "Architectural Blueprints—The '4+1' View Model of Software Architecture." *IEEE Software 12*, 6 (November 1995): 42-50.

[Lewis 2009]
Lewis, Bruce; Hugues, Jérôme; Wrage, Lutz; Feiler, Peter; & Morley, John. "Model-Based Verification of Security and Non-Functional Behavior using AADL." IEEE Security & Privacy, 2009.

[Lopez 2013]
Lopez Research. *An Introduction to the Internet of Things (IoT)*. http://www.cisco.com/web/solutions/trends/iot/introduction_to_IoT_november.pdf (November 2013).

[Mahaffey 2015]
Mahaffey, Kevin. “Hacking a Tesla Model S: What We Found and What We Learned.” https://blog.lookout.com/blog/2015/08/07/hacking-a-tesla/.

[Mattern 2010]
Mattern, Friedemann & Floerkemeier, Christian. *From the Internet of Computers to the Internet of Things*. http://vs.inf.ethz.ch/publ/papers/Internet-of-things.pdf (2010).

[McGraw 2010]
McGraw, Gary. “Software [In]security: How to p0wn a Control System with Stuxnet.” InformIT. http://www.informit.com/articles/article.aspx?p=1636983 (September 23, 2010).

[Miller 2015]
Miller, Charlie & Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. http://illmatics.com/Remote%20Car%20Hacking.pdf (August 10, 2015).

[Miller 2014]
Miller, Charlie & Chris Valasek. *A Survey of Remote Automotive Attack Surfaces*. http://blog.hackthecar.com/wp-content/uploads/2014/08/236073361-Survey-of-Remote-Attack-Surfaces.pdf (2014).

[Miller 2013]
Miller, Charlie & Chris Valasek. *Adventures in Automotive Networks and Control Units*. http://illmatics.com/car_hacking.pdf (2013).

[Mills 2010]
Mills, Elinor. “Stuxnet: Fact vs. Theory.” CNET News. October 5, 2010. http://news.cnet.com/8301-27080_3-20018530-245.html.

[MITRE 2015]
MITRE. *Common Weakness Enumeration*. https://cwe.mitre.org/ (2015).

[Nagaraju 2013]
Nagaraju, S.; Craioveanu, C.; Florio, E.; & Miller, M. *Software Vulnerability Exploitation Trends*. Microsoft, 2013. http://www.microsoft.com/en-us/download/details.aspx?id=39680.

[OMG 2013]
Object Management Group. *How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations*. http://www.omg.org/CISQ_compliant_IT_Systemsv.4-3.pdf (2013).

[Ouchani 2011]
Ouchani, S.; Jarraya, Y.; & Mohamed, O. A. “Model-Based Systems Security Quantification,” 142-149. *Proceedings from the 2011 9th Annual International Conference on Privacy, Security, and Trust*. Montreal, Quebec, Canada, July 19-21, 2011. IEEE, 2011.

[Riverbed 2015]
Riverbed Technology. OpNet Modeler. http://www.riverbed.com/products/steelcentral/opnet.html?redirect=opnet (2015).

[Roufa 2010]
Roufa, Ishtiaq; Miller, Rob; et al. "Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study," 21. *Proceedings of the 19th USENIX Conference on Security*. Washington, DC, 2010. USENIX Association, 2010.

[SAE 1996]
SAE. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (SAE ARP4761). SAE, December 1, 1996.

[Santucci 2014]
Santucci, Gerald. *The Internet of Things: Between the Revolution of the Internet and the Metamorphosis of Objects*. http://cordis.europa.eu/fp7/ict/enet/documents/publications/iot-between-the-internet-revolution.pdf (February 2010).

[SEI 2010]
Software Engineering Institute (SEI). Architecture Analysis and Design Language. http:www.aadl.info (2015).

[Shostack 2014]
Shostack, Adam. *Threat Modeling: Designing for Security*. John Wiley & Sons, 2014.

[Shostack 2012]
Shostack, Adam. *Elevation of Privilege: Drawing Developers into Threat Modeling*. https://www.usenix.org/system/files/conference/3gse14/3gse14-shostack.pdf (2012).

[Shostack 2010]
Shostack, Adam. *Elevation of Privilege: The Easy way to Threat Model*. https://www.youtube.com/watch?v=gZh5acJuNVg and https://www.youtube.com/watch?v=uDtVBoj9VpQ (September 17, 2010).

[Soni 1995]
Soni, D.; Nord, R. L.; & Hofmeister, C. "Software Architecture in Industrial Applications," 196-207. *Proceedings of the 17th International Conference on Software Engineering.* ACM, 1995.

[Vermesan 2013]
Vermesan, Ovidiu & Friess, Peter. *Internet of Things-Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013. http://www.internet-of-things-research.eu/pdf/Converging_Technologies_for_Smart_Environments_and_Integrated_Ecosystems_IERC_Book_Open_Access_2013.pdf.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2015	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Extending AADL for Security Design Assurance of Cyber-Physical Systems	FA8721-05-C-0003

6. AUTHOR(S)

Robert Ellison, PhD; Allen Householder; John Hudak; Rick Kazman, PhD; Carol Woody, PhD

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2015-TR-014

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Attacks such as the one that compromised the control systems for Iranian centrifuges demonstrate a growing need to improve the design of security in cyber-physical systems. While much of the work on security has focused on coding, many of the common weaknesses that lead to successful attacks are actually introduced by design. This technical report shows how important system-wide security properties can and must be described and validated at the architectural level. This is done through the adoption and use of the Architecture Analysis and Design Language (AADL) and a further extension of it to describe security properties. This report demonstrates the viability and limitations of this approach through an extended example that allows for specifying and analyzing the security properties of an automotive electronics system.

The report begins with a modeling of threats using the Microsoft STRIDE framework and then translates them into attack scenarios. Next, the report describes—as AADL components, relationships, and properties—the architectural structures, services, and properties needed to guard against such attacks. Finally, the report shows how these properties can be validated at design time using a model checker such as Resolute and discusses the limitations of this approach in addressing common security weaknesses.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Architectural description language, Architecture Analysis and Design Language, AADL, threat modeling, architectural security modeling, cyber-physical system security modeling, software assurance, Resolute, assurance case	72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102